李建江 薛巍 张武生 张为华 编著

并行计算机及编程基础

清华大学出版社

并行计算机及编程基础

李建江　薛　巍　张武生　张为华　编著

清华大学出版社
北　京

内 容 简 介

本书获"211 三期创新人才项目"资助，在参考国内外经典教材的基础上，结合新近出现的并行计算机体系结构与并行编程模型和语言，重点论述了并行计算基础、并行计算机体系结构、并行编程模型与语言、大规模稀疏线性方程组求解的并行化。主要内容：并行计算基础，包括：现实世界中的并行、并行与分布式计算的概念、来自应用领域的需求、并行程序设计的基本思想；并行计算机体系结构，包括：并行计算机传统体系结构及其比较与分析、多核 CPU 关键技术与未来发展趋势、GPU 与 GPU 集群的体系结构、Cell BE 关键技术及发展情况与典型实例、超级计算机等；并行编程模型与语言，包括：MPI、OpenMP、MapReduce、CUDA、Cell BE 上的编程模型与语言等；大规模稀疏线性方程组求解的并行化，包括稀疏线性方程组及其求解方法、大规模稀疏线性方程组求解案例、Helmholtz 方程计算的并行化、实际测试结果与性能优化。

本书主要面向从事高性能计算的程序员与工程师，使用并行计算机与并行技术加速专业领域计算的科研人员，以及对高性能计算感兴趣的程序员。开设相关课程的高等院校与科研机构也可选用本书作为教材或参考书。

图书在版编目(CIP)数据

并行计算机及编程基础/李建江等编著. —北京：清华大学出版社，2011.8
ISBN 978-7-302-26016-5

Ⅰ. ①并…　Ⅱ. ①李…　Ⅲ. ①并行计算机—计算机体系结构 ②并行程序—程序设计
Ⅳ. ①TP338.6 ②TP311.11

中国版本图书馆 CIP 数据核字(2011)第 120805 号

责任编辑：梁　颖　徐跃进
责任校对：焦丽丽

出版发行：清华大学出版社　　**地　　址：**北京清华大学学研大厦 A 座
http://www.tup.com.cn　　**邮　　编：**100084
社　总　机：010-62770175　　**邮　　购：**010-62786544
投稿与读者服务：010-62795954，jsjjc@tup.tsinghua.edu.cn
质　量　反　馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
经　　销：全国新华书店
开　　本：185×230　　**印　张：**15.25　　**字　　数：**313 千字
版　　次：20　年　月第 1 版　　**印　　次：**20　年　月第1 次印刷
印　　数：1～　00
定　　价：　.00 元

产品编号：

前　言 PREFACE

本书介绍了并行计算机体系结构、并行编程模型与语言等相关知识。本书内容共分为 4 章。第 1 章介绍并行计算的基础，包括并行计算的背景、并行编程模型、并行程序设计的基本思想；第 2 章重点介绍了并行计算机的体系结构，首先介绍了并行计算机传统体系结构的发展历程以及四种典型的并行计算机体系结构，然后对目前最新的基于多核(包括多核 CPU、GPU、Cell BE 等)的计算机体系结构进行详细介绍，有助于读者理解不同并行计算体系结构及其关键技术与未来发展趋势。第 3 章重点介绍了并行编程模型与语言，既包括目前应用最为广泛的 MPI 与 OpenMP，又包括最近出现的 MapReduce、CUDA、Cell BE 编程等知识，可帮助读者使用不同的并行编程模型与语言，在不同的并行计算机体系结构上进行并行程序的开发。第 4 章以大规模稀疏线性方程组求解的并行化为例，介绍了大规模稀疏线性方程组并行求解的全过程，有助于帮助读者将并行化思想应用于实际问题中。

北京科技大学计算机与通信工程学院李建江副教授完成全书的统稿，并与复旦大学计算机科学与技术学院张为华副教授编写了第 2、第 3 章，清华大学计算机科学与技术系薛巍副教授与张武生副教授分别编写了第 4 章和第 1 章。

本书适合作为高等院校计算机科学与技术学科各专业本科生、研究生的教材，也可作为有并行计算需求的相关专业研究生的参考书。

本书直接或间接地引用了许多专家和学者的文献或著作，在此向他们表示衷心的感谢。

本书获“211 三期创新人才项目”资助，在此特别感谢！

特别感谢北京科技大学计算机与通信工程学院计算机科学与技术系的崔健、曹伟、李明、吕义华、李福林、李远策、潘建、杨絮、姜涛、张磊、王聃、孙丽丽、方贤俊等同学以及清华大学计算机科学与技术系的李霖枫同学为本书的资料收集与实验操作所做的贡献。

清华大学出版社的编辑们为本书的出版做了大量的工作，在此对他们的辛勤工作与热情支持表示诚挚的感谢！特别感谢魏江江主任、梁颖和徐跃进老师给予的帮助！

由于作者水平有限，书中难免有疏漏和不妥之处，敬请读者批评和指正。

作　者

2011 年 4 月于北京

本书特色 This book special features

- 获“211 三期创新人才项目”资助。
- 保留传统的并行计算机体系结构（SMP、DSM、MPP、机群）及目前应用最为广泛的并行编程模型与语言（MPI、OpenMP）的相关内容。
- 重点论述最近出现的并行计算机体系结构（多核 CPU、GPU、Cell BE）及并行编程模型与语言（MapReduce、CUDA、Cell BE 编程）。
- 提供大量的实例，可操作性强。
- 适合作为高等院校计算机科学与技术学科各专业本科生、研究生的教材，也可作为有并行计算需求的相关专业研究生的参考书。

目录 CONTENTS

CHAPTER 1 第1章 并行计算基础

1.1 背景

在当代科研活动中，计算已经成为与理论、实验鼎足而立的第三个支柱。不断提高的计算能力为科技工作者提供了大规模数据处理分析和复杂理论模型研究的有效手段，使人们能够在更加深入、精细和更大规模的水平上对研究对象进行分析和计算模拟，成为发现新现象、认识科学规律、进行工程设计不可替代的手段。我国大气物理研究奠基人叶笃正院士曾在1985年指出，超级计算使得大气科学从一门经验科学变成了理论科学和实验科学，人们可以通过计算模拟来研究和预测天气和气候的变化。目前，人们用计算的方法研究从基本粒子、生命现象到宇宙演化、社会系统等各种问题。这极大地改变和拓展了传统的研究方法，推动了新学科领域的产生和科学研究能力的革命性发展，对当代科学和技术前沿的开拓起着不可替代的作用。

很难想象如果不借助超级计算技术，现今的天气预报将如何开展；而在药物设计领域，计算技术改变了药物筛选的模式，使其从原先的“体外(in vitro)筛选→体内(in vivo)筛选”变为“虚拟(in silico)筛选→体外(in vitro)筛选→体内(in vivo)”。超级计算机的发展更使其实现了虚拟筛选的高通量化，使得新药研发的周期缩短了0.9年，研发的直接费用降低达1.3亿美元。此外，计算模拟在新材料设计、新型纳米结构与分子器件设计、全球气候变化研究、工程设计、航空航天器制造等方面都发挥着重要作用。

世界上许多国家对计算能力的建设和计算科学的发展都给予了高度重视。美国从1970年起就实施了一系列推动计算科学发展的国家计划，包括“战略计算机计划”(SCP)、“高性能计算和通信计划”(HPCC)、“加速战略计算计划”(ASCI)、“先进计算设施伙伴计划”(PACI)等。2005年，在美国总统信息技术咨询委员会(PITAC)所做的报告《计算科学：确保美国的竞争力》中指出“二十一世纪最伟大的科学突破将是计算科学所获得的成就”，该报告建议“联邦政府、学术界和工业界必须共同制定一个数十年的发展蓝图，在科学和工程学科方面推动计算科学的发展”，并警告说：“美国现正处在关键时刻，如果我们还不高瞻远瞩和承担自己的义务，长此以往，国家的科学领导地位、经济竞争力和国家安全的后果不堪设想。”2006年，NSF提出了到2010年建设千万亿次计算规模的国家超级计算环境。

1.1.1 现实世界中的并行

计算的目的在于利用电子计算机系统通过人工建模的方式来模拟客观世界中事物及其之间的联系和运动规律。遗憾的是，到目前为止，能够构造的电子计算机系统均以一个串行模型(冯·诺依曼模型)为基础。但在真实的客观世界中，事物之间的联系不但复杂多样，而且其运动规律具有本质的并行特性。如何使用现有的串行模型来模拟并行的、具有复杂联系的客观物质世界一直是信息技术的核心。

使用电子计算机模拟客观物质世界的运转和状态不外乎两种手段：恰当的硬件架构设计和足够抽象的软件模型。下面，通过两个实例来说明该过程。

1. 实例1——组织结构

大多数信息系统(如Web服务、信息搜索、管理信息系统等)的主要(或部分)功能都可归结为事务处理。比如，一个办公自动化系统，要管理实际运行中的组织机构(包括人员、部门、权利等)、物理设施(如房产、工具、材料等)以及它们之间的交互、流通、与转换。这种系统的真正目的是利用计算机系统来建立一个有效的模型，模拟现实世界中组织机构运转的真实过程。

一个大的组织机构，可能下辖多个分支机构，每个分支机构又下辖若干个部门，其下可能还被划分成更多的人员分组。人员要在分组、部门、分支机构乃至整个大的组织内流动，变换不同角色。同一时刻，组织内要发生很多的事情，计算机要适时处理这些变化并将其呈现给对此感兴趣的目标。因此，需要将若干计算机通过网络互联起来(在其上运行软件系统)，对组织机构中的上述对象建立模型，管理各个对象的属性及其之间的交互，并最终通过网络将其映射到物理机器上执行。

现实社会的组织机构是为一定目标而设立并运行，其日常的运转过程就是完成一系列任务的过程。对于一个大的任务，人类社会实现它的方法是分而治之，即通过预定的管理手段和流程对任务进行分解，并将其分配给下属分支机构，最终经过层层分解，将具体的工作落实到个人——这在计算机系统中，分别对应于不同规模的执行单元。人员、分组、部门、分支机构等在执行任务的过程中要有信息交互，要互相协调。同样地，计算系统在处理一个大的问题时也要遵循这样的原则去操作。

比如，房地产公司要修建一栋大楼，其主要工作包括完成图纸设计并组织施工。完成该任务，不同组织(房地产公司)会有不同的做法，但不同的组织方法(管理机制)会在很大程度上影响对任务的进度和质量。一个好的组织者可能会将图纸设计和施工(准备)并行起来。在不同阶段内，还可通过任务分解和人员组织再进一步将具体工作进行并行实施。对难以同时推进的任务，有经验的管理人员还可通过合理的人员调配与进度安排创造出并行实施的条件。在任务并行过程中，管理人员还要使分配的任务之间具有足够的协调

机制，以确保子任务作为整体工作目标的有机组成部分。这些也都是并行计算中采用的基本做法。

综上所述，从本质上讲，信息处理（或称之为计算）过程具有天然的并行性。

2. 实例 2——科学计算

再看一个更加具体的例子。

我们知道，数值模拟是气候变化研究的重要手段之一。研究全球气候变化，需要综合考虑大气、陆地、海洋、冰川、植被、火山、人类社会活动等多种因素。环境中的这些要素每时每刻都在变化，而且需要使用不同的数学模型对它们分别进行描述。在真实的物理世界里，它们之间有着很自然的边界，并通过这些边界完成物质和能量的交换。如果通过计算机系统来模拟该过程，相应地需要在系统中对诸要素进行建模，并模拟各个要素之间的交互。在现实中，海洋、陆地、冰川每一时刻都在活动，一个子系统当前时刻的活动结果可能经过若干时间后，会被耦合到其他子系统中。因此，计算模型也要能体现这种并行推进的物理过程。

同时，为了提高计算速度，还可能进一步将一个子系统的计算过程分解，让多个计算系统同时工作以加快求解的速度，这里也利用了人类世界里将任务分而治之并统筹协调的思想。只不过，现实世界的分治协调是通过管理人员的经验与智慧，而计算机世界的并行计算则基于软件算法和程序架构的精巧设计。

由此可见，使用串行执行指令的电子计算系统来模拟并行运行的客观物质世界，关键在于建立一个有效的软硬件模型，并具备足够的描述能力来表达客观事物本身的运动变化以及事物之间的联系。软硬件模型需要相当的抽象和进化功能，以适应物理系统的发展变化。

在当今流行架构上的计算系统，普遍利用了软件平台的多任务机制来支持并行。具体来说，就是将计算指令流分成若干集合，让处理单元轮流对其执行。至于分解后的统一协调问题，则必须在软件设计阶段加以解决。在每个指令流集合内部，处理单元顺序执行，而在指令流集合之间则可实现并行执行。然而，现实世界的情况是：在每一时刻，该发生的事情必然会发生，其先后顺序由事物本身的物理规律决定，而不是遵循一个人为指定的序列。

因此，在这一点上，计算系统与现实世界的真实情况并不匹配。如何解决这个不匹配，是并行计算的重要任务同时也是难题之一。

1.1.2　并行与分布式计算的概念

信息领域总是不乏频繁出现的新概念、新定义，而且它们总是随需而变。在并行与分布式计算领域，有“分布式计算”、“并行计算”、“网络即计算机”、“元计算”、“网格计算”、

"云计算"等各种说法。关于这些,虽然没有一个概念的具体定义得到了全体认可,但每个概念都有一个基本为大家所公认的内涵。为澄清本书所讲述的内容,有必要对这些概念做一个简单的说明。

首先,并发(concurrent)计算、并行(parallel)计算、分布式(distributed)计算都是非常相似的概念,在实际系统中,它们之间只有一个模糊的界限。分布式系统最早可追溯到20世纪70年代以太网出现之际。比如,ARPANET与E-mail可以说是在当时最为成功的分布式应用。

并发的概念最早出现在操作系统中。实际上,是为了解决在串行执行指令的系统上运行多用户多任务的应用问题。随着网络的出现,并发的概念逐步扩展到由多台物理设备组成的分布式环境,进而催生了所谓分布式系统/分布式计算等概念。

通常来说,"分布式"依设备数据的布局而定,如果所有处理单元都能以共享的方式访问全局数据,则该系统为"共享内存"系统。而如果一个处理单元在本地只能访问到全体数据中的一部分,通过网络才能访问全体数据中的另一部分(通常在另外的物理设备上)数据,则称该系统为"分布式内存"系统。

并发、并行、分布式计算都是指在某一时刻同时有若干指令序列(或指令集合)在运行。在单处理器设备中,这种同时处理实际上依赖于操作系统的调度,通过轮转执行机制来实现。而在多核心/多处理器设备中,操作系统则可将不同的任务(指令序列)调度到不同的处理器(核心)上,以实现真正的同时执行。在多个设备通过网络连接起来形成更大规模的系统中运行的业务,即所谓的分布式处理,则是通过应用软件的自身机制来维护全局任务自身的语义。

从硬件环境角度来看,在分布式系统上,既可运行并行计算任务也可运行分布式计算任务。而共享内存系统上,通常运行并行任务或并发任务。借助于软件进行模型抽象,也可将一个共享内存系统视为逻辑上的一个分布式系统,进而将分布式计算任务运行在共享内存的物理设备上。此时,其通信手段不是借助网络而是直接使用内存空间。

在计算设备上运行的业务,通常有事务处理型和科学计算型。日常用到的信息处理、网络信息获取等均可归结为事务处理型应用。而科学计算类型的应用则纯粹利用处理单元的数值计算功能。这两者的主要区别在于:事务处理多以整数运算为主,兼借助于少量的简单浮点运算,而科学计算类型应用则主要以密集的高精度浮点运算为主。

在习惯上,认为分布式计算的内涵更广泛些,其既包括事务处理类型的业务,又包含科学计算类型的业务。而通常所说的并行计算,则一般只限于科学计算类型的业务。但这并不意味着任何一个分布式系统都适用于科学计算。科学计算类型的应用,多以密集的高精度浮点运算为主,一个任务往往由大量的迭代组成,对通信带宽、延迟、通信质量和效率的要求较高,因此一般要求底层的运行平台采用同构、紧耦合的系统。

分布式的事务处理型应用,其任务的划分与组织往往与其所描述的客观物质世界中

事物之间的联系密切相关，即其分布式建模可接受自然的基本需求的指导。而科学计算型应用，其软件模型来自于描述物质世界中运动变化的数学模型，多是一种(组)或若干种(组)复杂方程的数值求解。科学计算型应用中的任务分解往往由人为主观决定，因此模型的设计要更多考虑软件机制及其与底层硬件平台的结合情况。

从本质上讲，云计算、网格计算等实际上都是分布式计算，二者的目标都是为公众提供计算能力的基础设施，这里所提到的计算能力，实际上具有比科学计算更广泛的含义。科学计算可作为云计算或网格计算所提供的服务之一。同时，云计算或网格计算还可提供科学计算之外的服务，如事务处理、信息搜索等。

1.1.3　来自应用领域的需求

近年来软硬件技术的飞速发展，使得科学计算越来越成为科研与工业创新的重要手段之一。表 1-1 展示了 2010 年 11 月全球 TOP500 计算机应用领域的统计情况。可以看出，它们几乎涉及到社会生活的各个主要方面，不仅包含基础研究领域，在一些基础工业领域中，超级计算机的使用也为一些传统产业带来了更多、更快的创新机会。

表 1-1　2010 年 11 月全球 TOP500 计算机应用领域统计(来自 http://www.top500.org)

应用领域	数量	比例	处理器总数
空间应用	5	1.00%	31 936
机械制造	4	0.80%	27 056
测试	5	1.00%	50 528
生物	1	0.20%	8640
咨询服务	1	0.20%	6768
数据库	2	0.40%	12 216
防务	17	3.40%	307 296
电力	1	0.20%	5320
能源	14	2.80%	114 092
环境	1	0.20%	144 640
金融	43	8.60%	267 864
地球物理	19	3.80%	79 440
硬件设计	2	0.40%	12 744
信息服务	35	7.00%	222 348
信息处理	8	1.60%	83 584
生命科学	1	0.20%	18 176
医药	3	0.60%	20 000
媒体	1	0.20%	5936
科研	82	16.40%	2 115 546
软件开发	5	1.00%	32 688

续表

应用领域	数量	比例	处理器总数
电信	12	2.40%	80 428
气候变化	8	1.60%	159 204
互联网服务	4	0.80%	29 680
天气预报	2	0.40%	7040
半导体	2	0.40%	11 352
数字媒体	2	0.40%	9456
娱乐	2	0.40%	10 672
不确定应用	170	34.00%	2 261 923
零售商业	4	0.80%	25 968
服务业	44	8.80%	309 786
Totals	44	100%	6 472 327

1.2 并行编程模型

在高性能计算领域，软件环境与硬件平台密切相关，没有一个通用的平台适合高效解决所有种类的问题，也没有一种软件开发模型能够高效使用所有种类的硬件平台，对应用程序而言亦是如此。本小节将简要介绍并行编程模型。值得注意的是，并行编程模型与硬件平台的结构密切相关。

目前，在计算机系统上进行并行计算，编程模型主要有如下三种：

- 适用于共享内存的多线程编程模型；
- 适用于分布内存的消息传递编程模型；
- 混合编程模型。

1.2.1 适用于共享内存的多线程编程模型

① 操作系统库支持的多线程。通常情况下，都是使用库函数将应用程序的计算核心在独立的线程中进行实施。现代操作系统都支持多线程共享内存的并发，但只有在硬件的支持下才能实现真正的并行，可能的硬件环境包括：

- 支持超线程的单核 CPU（如 Intel Hyper-Threading 技术的 Pentium 4 或 Xeon 等）；
- 多核 CPU；
- SMP 系统；
- 上述三者的组合。

② 基于 OpenMP 的多线程。OpenMP 定义了一组编译指导语句、运行时库与环境

变量来影响并行程序的行为。

1.2.2　适用于分布内存的消息传递编程模型

消息传递模型适合分布式共享内存环境下的并行。常用的消息传递库为 PVM 和 MPI。其中，MPI 已被移植到多种平台上运行，包括 Linux、Windows、MacOS 等，也可在 SMP 系统上使用共享内存进行消息传递。可支持分布式环境下进程之间的点对点通信、单向通信、集合通信等。

1.2.3　混合编程模型

不管采用何种编程模型，最终都要解决全局与局部的关系。随着系统规模和计算规模越来越大，任何一种编程模型都不可能独立适用于所有的场合，因此，需要考虑使用混合编程模型。

目前，高性能计算系统平台的规模越来越大，千万亿次计算能力已成为现实，系统中集成的处理器数量也已达 10 万个以上，而且同一处理器内各个核心的结构也彼此不同。对于这样的计算机系统，则必须采用混合编程模型。即将消息传递编程模型与多线程编程模型相结合，实现多层次、可扩展的并行程序。

不管采用何种编程模型，由于在分布内存环境中存取本地资源要快于存取远程资源，因此为了获得高的执行性能都必须解决全局与局部的关系。即在程序设计之初，就要充分考虑目标平台的存储层次模型与网络拓扑结构，尽量将运行过程中动态的数据存取操作限制在结点/处理器/CPU 核心的“本地”，让最近即将使用的数据尽量填充在最接近 CPU 核心的存储器上，同时在次级存储器上将以后可能用到的数据准备就绪。

1.3　并行程序设计的基本思想

计算技术本身也是一个研究领域，它涉及计算数学、计算机系统结构、计算机软件以及各应用领域的相关知识。对各专业应用领域的研究/开发而言，计算技术作为一种工具来使用。因此，更应关注如何将本领域的计算模型转换为并行计算程序，所得到的计算软件应具备高性能、良好的规模可扩展性等特征。其中，规模可扩展性包括解决本领域计算问题本身的规模可扩展性与计算平台规模的可扩展性这两层含义。

在大规模并行计算领域，没有完全意义下的通用平台和通用软件，总是存在某种平台擅长某种类型的计算，某种应用程序在特定平台上才能获得最佳的效率。因此，开发并行应用程序，关键要做好如下三个阶段的工作：

- 理解用于计算的硬件平台结构，包括处理器体系结构、多核心的连接结构、内存层次结构、网络拓扑结构、I/O 系统结构等内容；

- 在理解所用硬件平台结构的基础上，理解系统软件提供的功能，并据此选择相应的编程模型；
- 寻找合适的工具/中间件来构建并行算法，最终实现并行应用程序。

构建并行应用程序，还应遵循如下几个原则。

1. 发掘并行性

从问题本身的特点出发，找出计算过程中可被并行执行的关键部分。有些问题天生具有良好的并行性，比如待处理数据集合可被划分为若干个互不耦合的空间(例如，石油勘探领域中的许多地震资料处理技术都具有这样的属性)，从而便于进行并行计算。但有些问题却没有明显的可并行特征，此时需要对求解这些问题的算法进行必要的改造以创造并行性。比较常用的办法有：

- 计算分解法，即通过公式推导与改造将原有的计算过程分解为若干个分步，减弱各分步之间的数据耦合，从而可在一定程度上(如采取流水线方式)进行并行执行；
- 循环调整法，即通过调整算法内的多重循环来改变计算的顺序与数据依赖性，从而使得部分计算任务可并行执行。

2. 保持高性能

一个将在并行计算机上计算的任务，必然会被分解——不论是按照数据划分方式还是任务划分方式——最终需要跨结点分布的处理器与内存空间协调起来共同完成。任务被分解之后形成在具体处理器/结点上运行的模块，从本地结点上看每个模块独立运行，但从问题整体来看，各结点上运行的模块需要相互协调。这要求在编写并行应用程序时，针对每个变量、每段计算代码，都要同时保持两种身份——局部和整体，所谓“既见树木又见森林”。局部模块要运行得尽量快，同时其作为全局的一部分，要兼顾与其他模块的协调。

一旦涉及到模块之间的协调，就会发生本地和远程的关系。在计算系统中，处理器永远都是访问离自己最近的存储空间时速度最快，处理器访问速度从快到慢的顺序依次为L1 cache→L2 cache→本地结点内存→远程结点内存和/或磁盘，这个存储层次的容量则恰好相反。因此，在组织数据时，并行应用程序需要精心安排数据在存储层次上的分布，以取得各级存储的高效利用。特别是涉及多核等共享缓存/内存的情况时，数据的存储分配策略将对并行应用程序的执行性能产生更大的影响。

此外，大规模数值计算程序往往都涉及大量的迭代过程，一个细微之处的时间消耗都可能会被放大到迭代次数倍。因此，要针对目标平台的体系结构细节来精心优化高阶循环内部的代码。

最后，数据局部性访问的特点决定了在算法设计时需要尽量减少模块之间的耦合，即减少计算过程中的通信次数及其通信量，因此需要对通信进行改进，将通信与计算进行重叠。特别是在多核系统中，可利用不同的核心分别完成计算与通信，达到更佳的效果。在通信时，还需注意并行应用程序中的通信逻辑拓扑与实际系统的网络物理拓扑之间的相互匹配。

3. 保持良好的可扩展性

可扩展性一方面要求并行应用程序本身能够对领域内各种规模的问题进行计算，另一方面也要求能够随着计算平台规模的扩大而相应获得理想的加速比，这二者之间有一定的关联，因此需要综合考虑。

需要避免计算过程中可能出现的瓶颈，比如任务划分要充分考虑负载均衡特别是动态负载平衡，"对等"思想是维护负载均衡和保持可扩展性的关键之一。所谓"对等"就是在算法设计时尽量避免使用 Master/Slave 和 Client/Server 等模式，这些工作模式往往容易在 Master 和 Server 等处形成性能瓶颈。设计"对等"任务分解的关键是发掘问题的同构性质，从逻辑上和物理上人为界定边界，使得各处理器的工作地位完全同等。

本章小结

首先，简要介绍了并行计算的背景，以现实世界中的组织结构与科学计算两个实例来阐述在现实世界中存在的天然并行性以及并行过程。然后，对并发计算、并行计算、分布式计算等基本概念进行了简单说明。通过 TOP500 发布的计算机应用领域统计数据，展示并行计算的应用需求。最后，介绍了并行编程模型与并行程序设计的基本思想。本章可使读者初步了解并行计算的基本知识。

参考文献

1 Jaroslaw Nieplocha, Robert J Harrison, Richard J Little_eld. Global arrays: a portable "shared-memory" programming model for distributed memory computers. Proceedings of the 1994 ACM/IEEE conference on Supercomputing. Los Alamitos: IEEE Computer Society, 1994. 340～349+816.

2 Ron Kalla, Balaram Sinharoy, Joel M Tendler. IBM Power5 Chip: a Dual-Core Multithreaded Processor. IEEE Micro, 2004, 24(2): 40～47.

3 Michael Bedford Taylor, Walter Lee, Jason Miller, et al. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. Proceedings of the International Symposium on Computer Architecture. Los Alamitos: IEEE Computer Society, 2004. 2～13.

4 Sanjeev Kumar, Christopher J Hughes, Anthony Nguyen. Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors. Comput. Archit. News, 2007, 35(2): 162～173.

5 Weirong Zhu, Vugranam C Sreedhar, Ziang Hu, et al. Synchronization State Bu_er: Supporting E_cient Fine-Grain Synchronization on Many-Core Architectures. In ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture, pages 35～45, 2007.

6 Gary Lakner and Carlos Sosa. Evolution of the IBM System Blue Gene Solution. IBM, 2007.

7 Roadrunner System Overview, Ken Koch, Roadrunner Technical Manager, Computer, Computational, and Statistical Sciences Division (CCS), http://www.lanl.gov/orgs/hpc/roadrunner/rrinfo/RR%20webPDFs/Roadrunner%20System%20Overview%20Oct%2017,%202007%20(LAUR).pdf.

8 Roadrunner Applications Team: Cell and Hybrid Results to Date, John Alexander Turner; Computer, Computational, and Statistical Sciences Division (CCS); Group Leader, CCS-2, Computational Physics and Methods, http://www.lanl.gov/orgs/hpc/roadrunner/rrinfo/RR%20webPDFs/Turner_Apps_v6_LA-UR.pdf.

9 Programming Roadrunner (and everything else), Paul Henning, Computational Physics Group (CCS-2); Computer, Computational, and Statistical Sciences Division (CCS), http://www.lanl.gov/orgs/hpc/roadrunner/rrinfo/RR%20webPDFs/Henning_Milagro_final1.pdf.

10 Performance of Roadrunner at Scale Under a Realistic Workload, Adolfy Hoisie, Leader, Computer Science for High-Performance Computing Group (CCS-1), Performance Architecture Lab (PAL), http://www.lanl.gov/orgs/hpc/roadrunner/rrinfo/RR%20webPDFs/RR%20Performance%20Talk%20PAL.pdf.

CHAPTER 2 第 2 章 并行计算机体系结构

2.1 并行计算机传统体系结构

并行计算机是相对于串行计算机而言的。在功能上,并行计算机比串行计算机更加强大;在设计方法和体系结构上,并行计算机比串行计算机更加复杂,需要考虑更多的问题。学习高性能计算编程必须对并行计算机的体系结构有一定的认识。

Flynn 分类法中的 SIMD 与 MIMD 计算机都属于并行计算机。其中,MIMD 计算机是并行计算机的主流和发展方向。本节将着重介绍几类最重要的 MIMD 计算机的体系结构。

2.1.1 共享存储与分布存储

不同类型的并行计算机实现多机并行工作的方式不同。其中,按照通信方式来划分,采用共享公共存储器中数据的方式来实现通信的并行计算机称为多处理机(multiprocessors),通过消息传递的方式来实现通信的并行计算机称为多计算机(multicomputers)。与之相对应,按照组织结构和存储方式来划分,可将并行计算机分为两种基本的类型:共享存储多处理机系统和分布存储多计算机系统。

共享存储就是具有所有处理器都能访问的物理内存,地址空间统一进行编码,各处理机采用读写内存中共享数据的方式进行交互的结构模型。在存储器硬件结构的实现方法上,又可将其分为集中式共享存储器和分布式共享存储器两种,前者由单一的存储器构成,后者由统一地址空间编码的多个存储器构成。共享存储的多处理机是一种紧耦合型的系统,其优点在于通信机制简单,使得程序开发更加简便,可移植性更好。但是,由于共享访问存储介质,处理器的访存过程需要经历竞争和延迟,这将在一定程度上制约共享存储多处理机的速度。

分布存储是多个处理机拥有自己独立的存储器,彼此之间不共享,处理机之间通过互联网络连接,以消息传递的方式实现通信的结构模型。分布存储多计算机系统可以采用松耦合的连接方式,灵活性比较好,可扩展性好,能够完成大计算量的任务,同时访存压力比较小。但是,分布存储多计算机系统的结构比较复杂,通信和负载均衡都需要程序开发者来安排,因此,在其上进行程序设计的难度比较高。

一般来说,共享存储的多处理机优势为程序开发简单,一致性较好;分布存储多计算

机系统优势为可扩展性好，性能较高。对于应用人员和系统设计人员来说，应根据自己的需求特点来选择合适的计算机体系结构。

2.1.2 并行计算机传统体系结构的发展

20 世纪 60 年代，随着计算机硬件技术的发展和应用需求的不断增长，开始出现并行计算机的萌芽。20 世纪 60 年代初出现的 CDC6600 就采用了双 CPU 连接多个外部处理器的结构，它已具备了并行计算机的硬件特点。20 世纪 60 年代后期开始出现的指令级流水工作方法进一步推动了并行计算机的出现。

1972 年，Illinois 大学和 Burroughs 公司联合研制出世界上第一台真正意义的并行计算机 ILLIAC IV，它是一台具有 32 个处理单元的 SIMD 类型的计算机，采用环状连接拓扑结构，用于流体力学方面的运算，迈出了并行计算机研制的第一步。20 世纪 70 年代诞生的并行机还有阵列机 ICLDAP、Goodyear MPP 以及向量机 CRAY-1、STAR-100 等，它们都属于 SIMD 类型的计算机。其中，向量机 CRAY-1 获得了很好的向量计算效果。从 20 世纪 70 年代开始，并行计算机逐渐引起人们的极大兴趣，吸引了大量的专家学者致力于并行计算机的研制和并行程序的设计，为 20 世纪 80 年代并行计算机的蓬勃发展奠定了坚实的基础。

20 世纪 80 年代早期，以 MIMD 计算机的研制为主。首先诞生的是 Denelcor HEP，它有 16 台处理机，采用共享存储的方式，能同时支持细粒度和粗粒度并行，并被应用于实际的计算中，使得许多人学会了并行计算。其次，诞生了共享存储向量多处理机 CRAY X-MP/22(2 个结点)与 IBM 3090(6 个结点)，它们都取得了很好的实际并行计算性能。同时，以超立方体结构连接的分布式存储 MIMD 原型机开始出现。

20 世纪 80 年代中期，共享存储多处理机系统得到了稳定发展。两台成功的机器为 Sequent(20 个结点)与 Encore(16～32 个结点)，它们提供稳定的 UNIX 操作系统，实现用户之间的分时共享，对当时的 VAX 系列串行机构成了严重的威胁。同时，还诞生了 8 个结点的向量多处理机 Alliant(它提供较强的自动向量并行编译技术)与 4 个结点的向量多处理机 CRAY-2。这些向量多处理机系统在实际应用中均取得了巨大的成功。与此同时，人们对共享存储多处理机系统的内存访问瓶颈问题有了比较清醒的认识，纷纷寻求解决办法以确保它们的可扩展性。在此期间，还诞生了可扩展的分布存储 MIMD MPP *n* 立方体(*n*CUBE)，该机器有 1024 个结点，CPU 和存储单元均分别包含在结点内，所有结点之间通过超立方体网络进行相互连接，支持消息传递的并行编程环境，并被投入实际的应用中。由于该机对流体力学中的几个实际应用问题获得了超过 1000 倍的并行加速比，引起了计算机界的轰动，改变了人们对 Amdahl 定律的认识，打消了人们对并行计算技术的疑虑。当时，在分布存储体系结构中，处理机之间的消息传递效率与消息的长度、处理机之间的距离有着较大的关系。因此，在互联网络最优拓扑连接和数据包路由选择算法

等方面的研究引起了人们的大量关注，其目的在于减少处理机远端访问的开销。

20 世纪 80 年代后期，真正具有强大计算能力的并行计算机开始出现。例如，Meiko 系统，由 400 个 T800 Transputer 通过二维网格(Mesh)相互连接构成，适合中粒度的并行。又如，CM-2、MasPar 与 DAP 这三台 SIMD 并行计算机。其中，CM-2 的 Linpack 测试获得了 5.2GFLOPS 的性能。通过超立方体(SuperCUBE)连接的分布存储 MIMD 并行计算机 *n*CUBE-2 与 Intel iPSC/860，可分别扩展至 8000 个结点和 128 个结点，峰值性能分别达到 27GFLOPS 与 7GFLOPS。由硬件支持共享存储机制的 BBN TC2000，使用 Butterfly 多级互联网连接处理机和存储模块，可扩展至 500 台处理机，本地 cache、内存和远端内存访问的延迟时间比例为 1∶3∶7。共享存储向量多处理机系统 CRAY Y-MP，能够获得很好的实际运算性能。

20 世纪 90 年代，得益于微电子技术的发展，基于 RISC 指令系统的微处理芯片几乎以性能每 18 个月增长 1 倍、内存容量每年增长 1 倍的速度发展。同时，网络通信技术也得到了快速增长，它们对并行计算机的发展均产生了重要的影响。在这个时期，MIMD 类型的计算机占据了绝对的主导地位，用于科学与工程计算的 SIMD 类型的计算机和单纯的向量机已逐渐退出历史舞台。考虑到共享存储并行机不可避免的内存访问瓶颈问题，人们纷纷把目光转移到分布式存储 MPP 系统，使得 MPP 的硬件和软件系统得到了长足的发展。由于微处理芯片性能和网络技术的发展，MPP 并行机大量采用商用微处理芯片作为单结点，通过高性能的互联网连接而成。由于普遍采用虫孔(wormhole)路由选择算法，使得消息传递时间不再与它所经过的结点数目相关，即处理机之间的消息传递开销不再与距离有关，或者相关的程度可以忽略不计，互联网的拓扑结构趋于统一。分布式存储并行程序设计以消息传递为主，少量的也支持数据并行，比如高性能 Fortran (HPF)。在该时期，为了让共享存储并行机具有可扩展性以适用于高性能计算，并继承共享存储并行机易于并行程序设计的优点，分布共享存储的思想已被人们广泛接受。这方面的代表机型为 1991 年生产的 Kendall Square KSR-1，它提供给用户透明的共享存储结构，每个环含有 32 个结点，多个环之间以层次结构相互连接，可扩展至 1024 个结点，峰值速度为 15GFLOPS。

20 世纪 90 年代中期，微处理器的性能已非常强大，能够提供每秒几亿到十几亿次的浮点运算速度。同时，互联网的点对点通信能力已达到每秒超过 500MB 的带宽。高性能微处理器和网络通信技术为并行计算硬件环境带来了新的面貌，使它们呈现出如下发展趋势：

- 以高性能微处理芯片和互联网通信技术为基础，共享存储对称多处理机(SMP)系统得到了迅速发展。它们大多以高性能服务器的方式出现，能提供每秒几百亿次的浮点运算能力、几十 GB 的内存和超过 10GB/s 的访存带宽。具有丰富的系统软件和应用软件，很强的容错能力、I/O 能力、吞吐量、分时共享能力和稳定性，友

好的共享存储并行程序设计方式，易于使用的并行调试与性能分析工具，为大量中小规模科学与工程计算、事务处理、数据库管理部门所欢迎。因此，它们一出现，就迅速抢占了原本属于共享存储向量并行机的市场，成为几百亿次以下并行计算机的主导机型。但是，它们仍然存在可扩展性较差的缺陷，不能满足超大规模并行计算的要求。

- 以微处理芯片为核心的工作站能提供近 1GFLOPS 的计算速度与几十 MB 的内存，能单独承担一定的计算任务。将多台这样的同构或异构型工作站通过高速局域网相互连接起来，再配备一定的并行支撑软件，形成一个松耦合的、协同地并行求解同一个问题的并行计算环境，称之为机群系统。由于机群系统具有投资风险小、结构灵活、可扩展性强、软件财富可继承、通用性好、异构能力强等较多优点而被大量的中、小型计算用户和科研院校所接受，成为高性能计算领域的一个新的发展热点，占据了原本属于传统并行计算机的部分市场。但是，它们仍然具有结构不稳定、并行支撑软件较少、并行开销大、通信带宽低、负载不均衡和并行程序设计难等许多亟待解决的问题，在当时吸引了大量国内外专家学者的注意力。
- 由于分布存储的并行计算机具有并行程序设计难、不易被用户接受的缺点，单纯的分布存储并行机已经朝着分布共享多处理机(DSM)方向发展。它们都采用最先进的微处理芯片作为处理单元，单元内配备有较大的局部 cache 和局部内存，所有局部内存都能实现全局共享，所有结点通过高性能网络相互连接，用户可以采用共享存储或数据并行的并行程序设计方式，并且可自由地申请结点数目和内存大小。

2000 年以来，受大规模计算(如天气预报、石油勘探)需求的牵引以及微处理器和商用高速互联网持续发展的影响，高性能并行计算机得到前所未有的发展。SMP、DSM、机群等各类并行计算机都得到了长足的进步。在低端市场上，SMP 以其良好的性价比逐渐替代了 MPP。在大型机领域，机群系统则以其结构灵活、通用性好、异构能力强等诸多优点逐渐代替 MPP 成为主流。并且，随着网络技术的发展，机群系统与 MPP 系统之间的界限变得越来越模糊。例如，IBM 公司的 SP2 系统既看成是 MPP，又可看作是机群系统。这体现了新世纪以来高性能计算机领域中体系结构更加灵活、逐渐融合的趋势。在 2001 年的全球高性能计算机 Top500 排名中，MPP 有 314 台，机群系统只有 32 台。但到了 2010 年，在 Top500 排名中，MPP 只有 74 台，而机群系统则达到了 424 台。这说明机群系统已成为大规模并行计算机系统的主要架构模式，而 MPP 正慢慢衰落。

传统并行计算机的历史发展如图 2-1 所示。

总的来说，SMP、DSM、MPP、机群系统这四种类型的计算机是最近 20 年中并行计算机的主要类型，也是最经典的并行计算机体系结构模式。下面，将分别介绍这四类并行计算机。

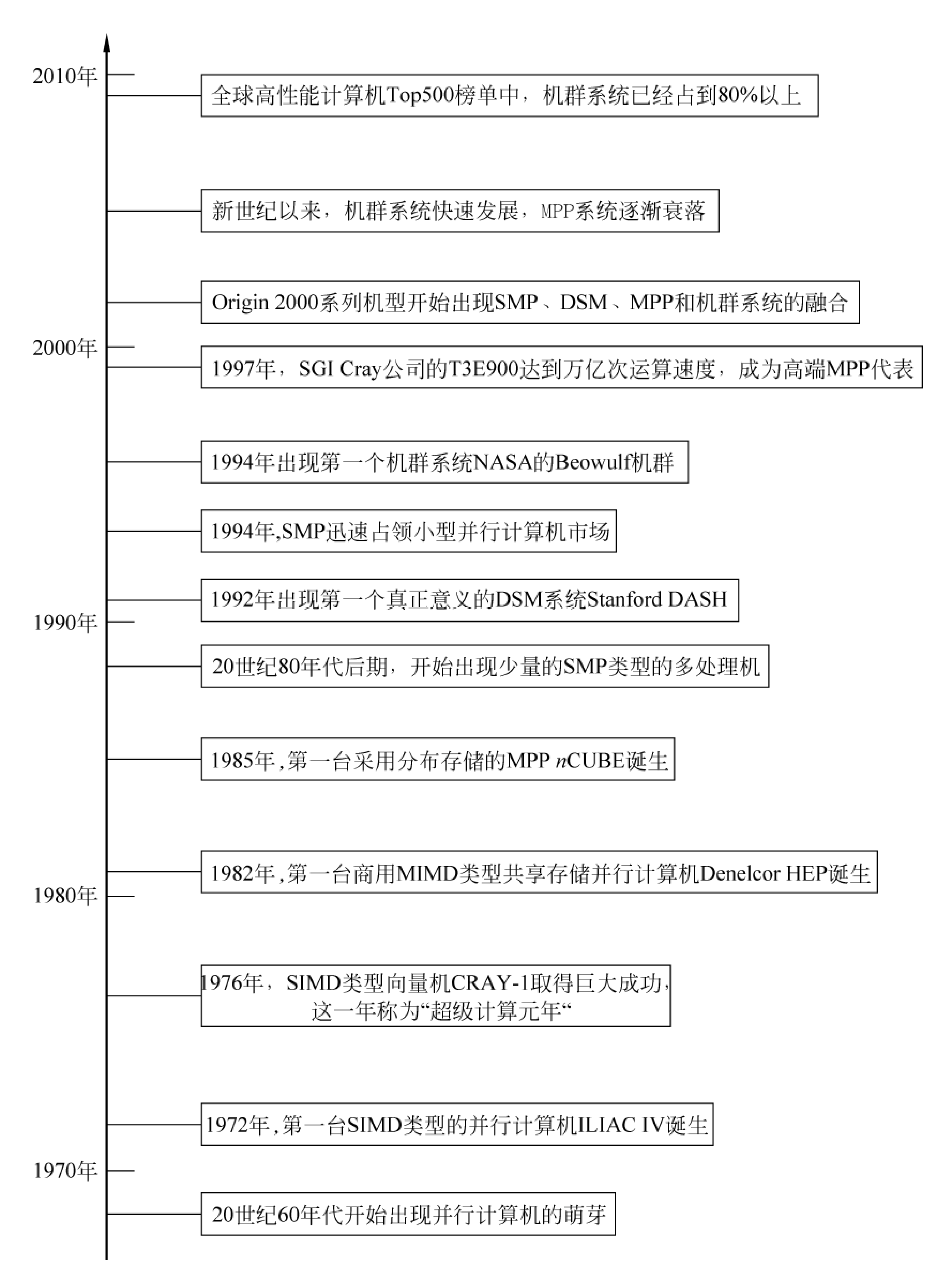


图 2-1 并行计算机发展历史图

2.1.3 SMP对称式共享存储器多处理机

1. 简介

对称多处理机(Symmetric Shared-memory Multiprocessor,SMP)是一类常见的共享存储并行计算机系统。一般来说,它的处理器数量比较少,各处理器共享一个集中式的物理存储器,各处理器的关系是对称的。在20世纪80年代,随着cache技术的发展,单个处理器对内存带宽的要求降低,人们开始设计通过总线来共享一个单独的物理存储器的小规模多处理机系统,即对称多处理机(SMP)。到20世纪90年代中期,SMP已经成为一种主流的并行计算机。其基本结构如图2-2所示。

2. 特点

SMP系统具有如下特点:

- 对称性,系统中各个处理器的结构相同,可以对称访问任意存储器和I/O设备;
- 共享性,采用共享存储方式,所有存储器的存储单元具有统一的地址空间编码;
- 低延迟,由于采用了共享存储方式,处理机之间的通信均由简单的读、写指令来完成,通信延迟比较低,通信开销小;

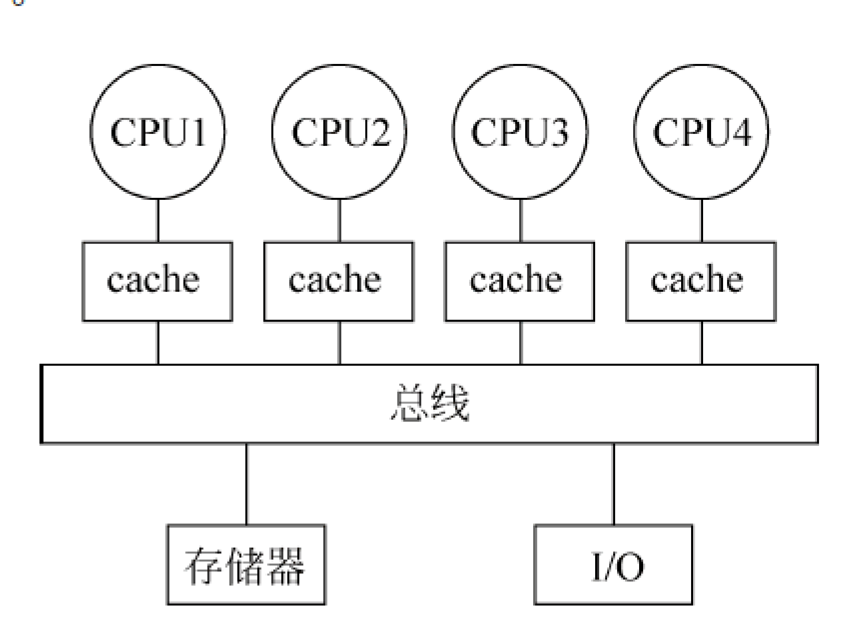


图2-2 SMP结构图

- 负载均衡,只有一个OS副本驻留在共享存储器中,OS根据负载进行进程调度,易于达到动态负载平衡。

SMP系统具有一定的局限性,主要体现在两个方面:

- 可扩展性较差,一般情况下SMP系统的处理机数目在8～16个之间,很难扩展到100个以上的处理机,这决定了SMP无法用于需要完成巨大工作量的情况;
- 可用性较差,所有处理机共享一套存储器和操作系统,一旦存储器或者操作系统出现问题,整个系统都将瘫痪。

因此SMP适用于运算规模较小并且对可用性要求较低的情况,在小规模并行计算机市场上它具有较高的性价比。

3. 关键技术

SMP系统要解决的一个主要技术问题就是cache一致性问题,造成cache一致性问题的原因有以下几点:

- 由共享可写数据造成的不一致;

- 由绕过 cache 的 I/O 操作造成的不一致；
- 由进程迁移所造成的不一致。

下面是几种解决 cache 一致性的协议和策略。

1）基于监听的 cache 一致性协议

基于监听的 cache 一致性协议的基本原理是：当某个 cache 需要访问存储器时，它会把请求放到总线上广播出去，其他 cache 控制器通过监听总线来判断它们是否有总线上请求的数据块，如果有则进行相应的操作。

通常使用写无效和写更新这两种策略来解决 cache 一致性问题。

- 写无效策略（write invalidate）：是指当某个处理器更新其私有 cache 中的某个数据时，它通知所有其他 cache 该数据在它们中的副本从此均无效，这样就可以避免其他“过时”的副本被使用而造成错误。
- 写更新策略（write update）：是指当某个处理器更新其私有 cache 中的某个数据时，它把所更新的数据发送给所有其他 cache，以更新这一数据在其他 cache 中的所有副本。

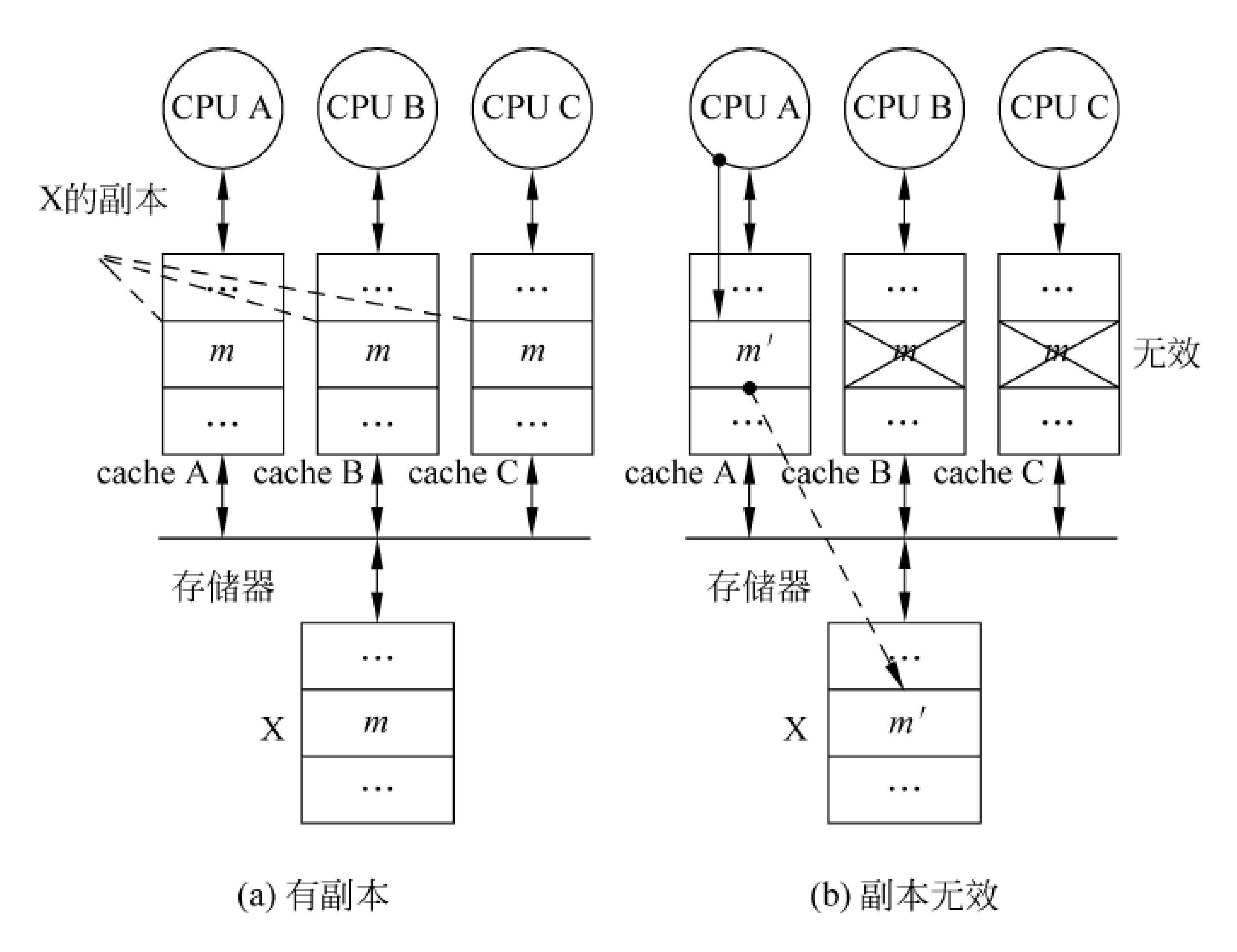
一般来说，使用写更新策略，需要传输更新后的数据，而写无效只需传输写无效信息。因此，写更新传输的数据量比写无效要大，而且，被更新数据的某些副本以后也不一定会被再次使用。

这里需要注意的是，写无效和写更新是维护处理器与 cache 一致性的策略，它与维护 cache 与主存储器一致性的策略没有必然的关系。SMP 系统还需要考虑 cache 与主存储器的一致性。维护 cache 与存储器的一致性有如下方法。

- 写回法：当 CPU 写 cache 命中时，只修改 cache 的内容，而不立即写入主存；只有当此行被替换时才写回主存。
- 写直达：又称全写法。当 CPU 写 cache 命中时，cache 与主存同时发生写修改，因而较好地维护了 cache 与主存内容的一致性。
- 写一次法：是写回法与写直达的折中方法。当 CPU 写 cache 命中与未命中时，处理方法与写回法基本相同，只是第一次写 cache 命中时要同时写入主存。这是因为，第一次写 cache 时，CPU 要在总线上启动一个存储写周期，其他 cache 监听到此主存块地址及写信号后，即可拷贝该块或及时作废，以便维护系统全部 cache 的一致性。

在实际操作过程中，需要从上述两套方法中各取一种来实现 cache 的一致性。例如，采用写无效策略和写直达策略（如图 2-3 所示）。处理器 A、B、C 的 cache 中都有数据 m 的副本（如图 2-3（a）所示）。当处理器 A 修改私有 cache 中 m 时，不但要向其他处理器的 cache 发送无效信息，而且要将共享存储器中该数据的副本更新。最终，处理器 A 的私有 cache 和共享存储器中的数据 m' 是相同而且是正确的，而 B 和 C 处理器的 cache 中该数

据的副本被标记为无效(如图 2-3(b)所示)。



(a) 有副本 (b) 副本无效

图 2-3 写无效与写直达策略实现一致性

2）基于目录的 cache 一致性协议

基于目录的 cache 一致性协议的基本原理是：使用 cache 目录来存放有关数据块拷贝驻留在 cache 中的信息，只把使其他 cache 数据块无效的一致性命令发送给存放有相应数据块的 cache，从而保证 cache 的一致性。

根据目录结构的特点，可将基于目录的 cache 一致性协议分为：基于全映射(full-map)目录的 cache 一致性协议、基于有限(limited)目录的 cache 一致性协议和基于链式(chained)目录的 cache 一致性协议。

- 全映射目录：每一个目录项都包含一个 N 位的位向量，其中的每一位对应一台处理机。其特点是处理比较简单，速度比较快。但是，存储开销很大，可扩展性较差。
- 有限目录：是对全映像目录的改进。采用位数固定的目录项，通过限制同一数据所在 cache 中的副本总数来实现。它克服了全映射目录的不足，但其缺点也非常明显，就是当同一数据的实际副本数目大于限量时，必须进行特殊的处理。
- 链式目录：是用一个目录指针链表来表示共享集合。这样就能在不限制同一数据所在 cache 中的副本数目的情况下保持可扩展性。

4. 典型实例

下面以原 Sun 公司的 T1 系统为例进行详细介绍。

T1 是 2005 年发布的一款作为服务器的多处理机。它采用了多线程与多核技术，其线程级并行性良好，吞吐率较高。每个 T1 多处理机有 8 个处理器，每个处理器最多支持 4 个线程，具有一条 6 段单流出流水线。T1 处理器主要属性如表 2-1 所示。

表 2-1　T1 处理器的主要属性

机　型	Sun T1
处理器情况	8 个相同处理器，共享一个浮点运算部件
多线程支持	每个处理器支持 4 线程，细粒度线程调度
一级 cache	16KB 指令 cache，8KB 数据 cache，64B 块大小，无竞争状态下不命中开销 23 个时钟周期
二级 cache	4 个独立二级 cache，每个 750KB，与存储器相连，64B 块大小，无竞争状态下，不命中开销 110 个时钟周期
初始版本工艺	90nm 工艺，最高时钟频率 1.2GHz，电源功率 79W，3×10^8 个晶体管，圆片面积 379mm^2

T1 的每个处理器带有 1 个 L1 cache，8 个处理器通过交叉开关与 4 个 L2 cache 相连。使用基于目录的 cache 一致性协议来实现 L2 cache 一致性，其目录表中对于每一个 L2 cache 块都有对应的项。通过把每个 L2 cache 与 1 个存储器相连，T1 实现了将目录表放在 L2 cache 而不是放在主存上，从而有效地降低了访存开销。L1 cache 采用写直达法，而且所访问的数据都能从 L2 cache 中获得。T1 处理器的结构如图 2-4 所示。

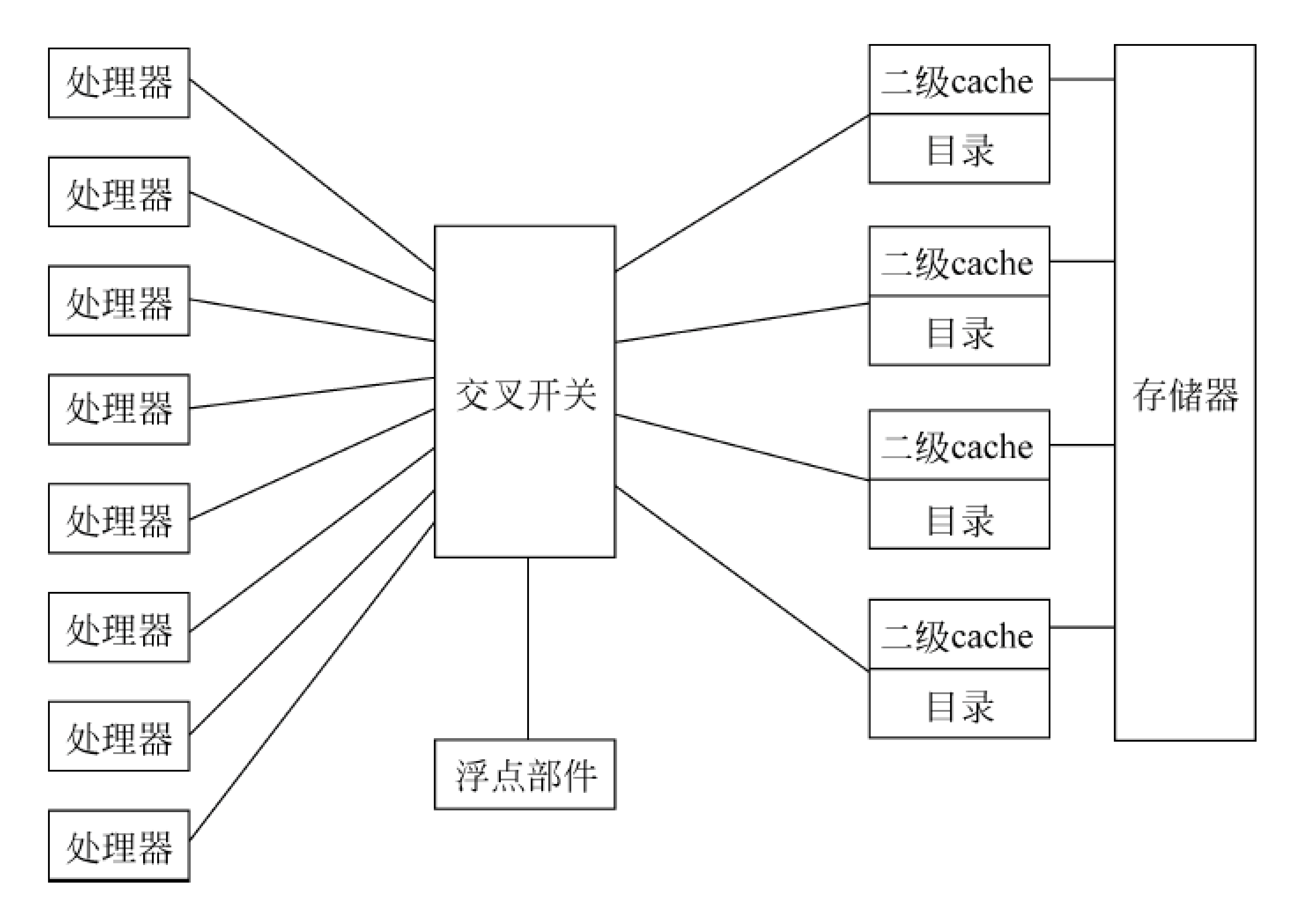


图 2-4　T1 处理器组成结构图

2.1.4 DSM分布共享存储多处理机

1. 简介

分布共享存储多处理机系统(Distributed Shared-Memory,DSM)是一种物理存储器分布于各处理结点,而逻辑地址空间采用统一编址的计算机。

在传统的共享存储器多处理机系统中,一般使用总线或交叉开关来连接共享存储器。然而,随着处理器规模的扩大和访存次数的增加,集中式的存储器成为系统的瓶颈。在这种情况下,提出了分布共享存储多处理机系统 DSM。DSM 就是在硬件上实际分布存储的系统上逻辑实现共享存储的模型。其结构如图 2-5 所示。

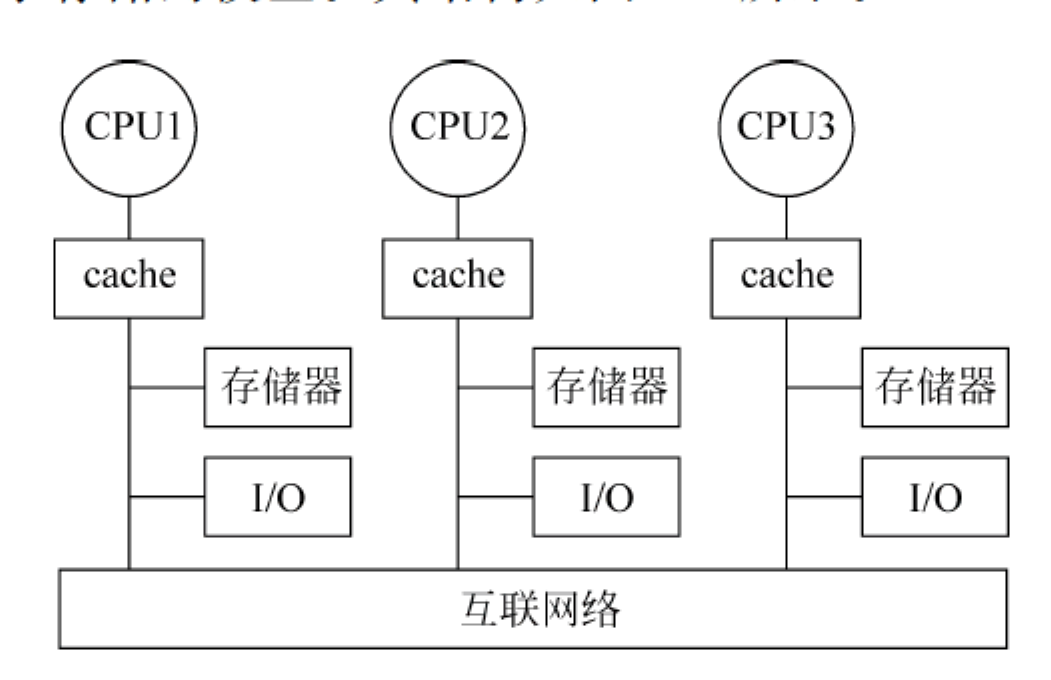


图 2-5 DSM 结构图

DSM 支持统一地址编程空间,从而有效地将传统的共享存储多处理机系统和分布存储多计算机系统的优点结合起来,兼具可编程性好和可扩展性高的优势。

2. 特点

DSM 同时具有共享存储和分布存储的特征,因此具有很多二者的优势,同时又避免了二者的一些弊端,是一种取长补短的系统。一般来说,分布共享存储多处理机系统 DSM 具有如下特点:

- 通用性,DSM 采用单地址编程空间,减轻了程序员的负担,具有较强的通用性;
- 可扩展性,DSM 的物理存储器分布在不同的位置,这使得系统能够支持更多的存储器,从而支持更大规模的并行计算机系统;
- 虚拟化,对上层用户屏蔽了处理器与非本地存储器之间的网络连接情况,提高了系统的可移植性;
- 访存时间受互联网带宽的影响较大,如果处理器访问的资源存储在本地,访存速度较快;如果处理器访问的资源存储在其他位置,访存速度则受限于网络带宽。

3. 关键技术

共享存储系统都采用 cache 来减少由共享导致的冲突和延迟对性能的影响。然而，由于分布共享存储系统中的存储器是分散的，不同处理器访问统一存储单元将会有不同的延迟。该问题称为非一致访存问题（Non-Uniform Memory Access，NUMA）。同时，DSM 系统也需要解决 cache 一致性问题，即如何保证同一数据单元在不同 cache 中的备份数据的一致性。

非一致性访存问题将带来访存时间的不一致，cache 一致性问题会带来同一数据单元的多个备份。这些问题破坏了存储访问的不可分割性（atomicity），使得同一数据单元在不同时刻被不同的处理器所访问，从而影响系统的正确性。为了保证正确性，需要对访存操作的发生次序进行严格的限制。许多在单处理机中行之有效的提高性能的技术（诸如，预取、多发射等）都不能在 DSM 中盲目使用，否则会影响系统的性能。可见，分布共享存储多处理机 DSM 具有独特的优势，同时也需要面对一些新的问题。

一般来说，根据存储器的组织方式和 cache 一致性的实现方法等特征，可将常见的分布共享存储多处理机系统分为如下几类：

(1) 高速缓存一致的非均匀存储访问结构 CC-NUMA。这类结构的共享存储器分布于各结点之中，结点之间通过互联网相连。每个处理器都能缓存共享单元，通常采用基于目录的方法来维护处理器之间的 cache 一致性。cache 一致性的维护是这类系统的关键，决定着系统的可扩展性。典型的例子有 Stanford 大学的 DASH 和 FLASH，MIT 的 Alewife，以及 SGI 的 Origin 2000 等。

(2) 高速缓存不一致的非均匀存储访问结构 NCC-NUMA。这类结构的每个处理器都有高速缓存，但硬件不负责维护 cache 一致性。cache 一致性由编译器或程序员来维护。在 Cray 公司的 T3D 和 T3E 中，系统为用户提供了一些用于同步的库函数，便于用户通过设置临界区等手段来维护数据一致性。这类结构的好处是系统可扩展性强，高档的 T3D 及 T3E 产品可达上千个处理器。典型实例是 Cray 公司的 T3D 及 T3E 系列产品。

(3) 唯高速缓存存储访问结构 COMA。这类结构的共享存储器的地址是活动的，存储单元与物理地址相分离。根据访存模式，数据可以在各结点的存储器之间动态地移动和复制。每个结点的存储器相当于一个大容量高速缓存，数据一致性也在这一级维护。这类结构的优点是在本地共享存储器命中的概率较高；其缺点是当处理器的访问在本结点不命中时，由于存储器的地址是活动的，需要一种机制来查找被访问单元的当前位置，因此延迟很大。典型实例有 Kendall Square Research 的 KSR1 和瑞典计算机研究院的 DDM。

(4) 共享虚拟存储访问结构 SVM，又称软 DSM 系统。在基于消息传递的 MPP 或机

群系统中，SVM 系统用软件的方法把分布于各结点的多个独立编址的存储器组织成为一个统一编址的共享存储空间。其优点是在消息传递系统上实现了共享存储的编程界面，但主要的问题是难以获得满意的性能。与硬件共享存储系统相比，SVM 系统较大的通信开销和共享粒度(通常是存储页，页大小由操作系统决定)会导致假共享及额外的通信。在基于机群的 SVM 系统中，通信开销会很大。与消息传递系统(如 MPI)相比，基于 SVM 系统的并行程序通信量通常会更大。

由于 SVM 结构的出现，使得采用消息传递模式的并行计算机系统也能虚拟实现 DSM。这也说明各种结构的并行计算机的融合是未来发展的趋势。

4. 典型实例

下面以 Stanford 大学研制的 DASH 并行计算机系统为例进行详细说明。

DASH 是 Stanford 大学在 1992 年研制的分布共享存储多处理机系统。DASH 是共享存储的目录结构(Directory Architecture for Shared Memory)的英文缩写。它采用了 CC-NUMA 结构，使用基于目录的方法维持 cache 一致性，具有分布式存储器和单地址空间，为建立具有单一地址空间的可扩展并行计算机提供了设计范例。DASH 的结构如图 2-6 所示。

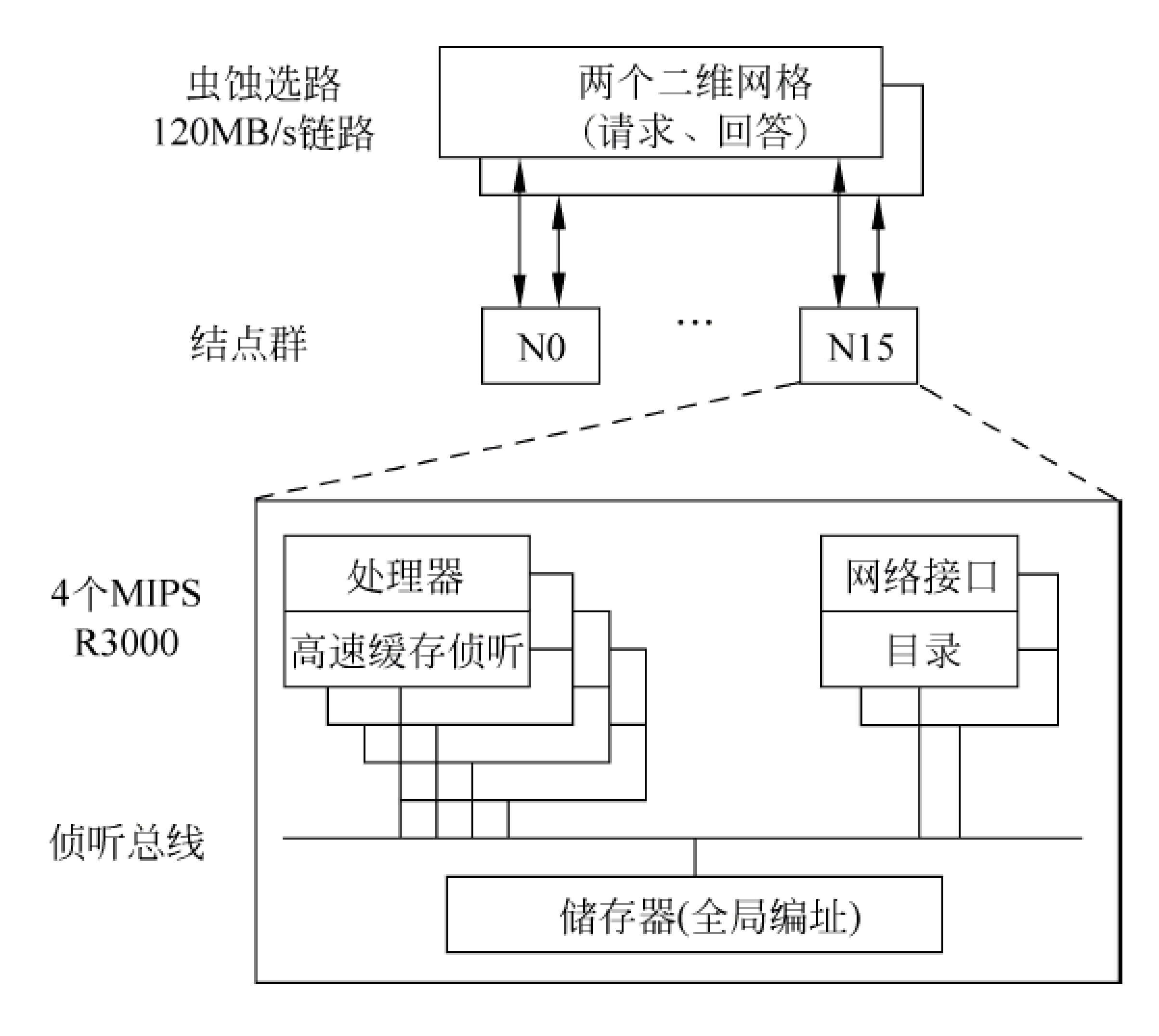


图 2-6 DASH 结构图

DASH 系统包含 16 个 SGI SMP 结点，每个结点有 4 个 MIPS R3000/R3010 处理器，总共有 64 个处理器，处理器频率为 33MHz。与一般 SGI SMP 结点不同的是，DASH 系统中的 SMP 结点有两块特殊子板，其上装有网络接口电路和 cache 目录。16 个 SGI

SMP 结点由采用虫蚀寻径方式的两个二维网格型网络(包括一个用于向远程存储器发送请求的请求网格和一个对应的应答网格)进行连接。网格的通道带宽为 16 位,通过时间为 50ns。

网格型网络可支持本地和全局存储器的频宽扩展,这有利于开发局部性。共享存储器的单地址空间更利于编译和程序设计。

在 1996 年,SGI 公司将 DASH 结构商业化,推出了 Origin 系列产品,称为可扩展共享存储器结构(Scalable Shared Memory Architecture,S2MA),从而使 DASH 成为 DSM 系统的标志性范例。

2.1.5 MPP 大规模并行处理机系统

1. 简介

大规模并行处理机(Massively Parallel Processor,MPP)是指由几百或几千台处理机组成的大规模并行计算机系统。MPP 系统中处理器数目巨大,整个系统规模庞大,许多硬件设备是专门设计制造的,开发起来比较困难,通常被视为国家综合实力的象征。同时,MPP 能够提供 SMP、DSM 等并行计算机不能达到的计算能力。MPP 的一般结构如图 2-7 所示。

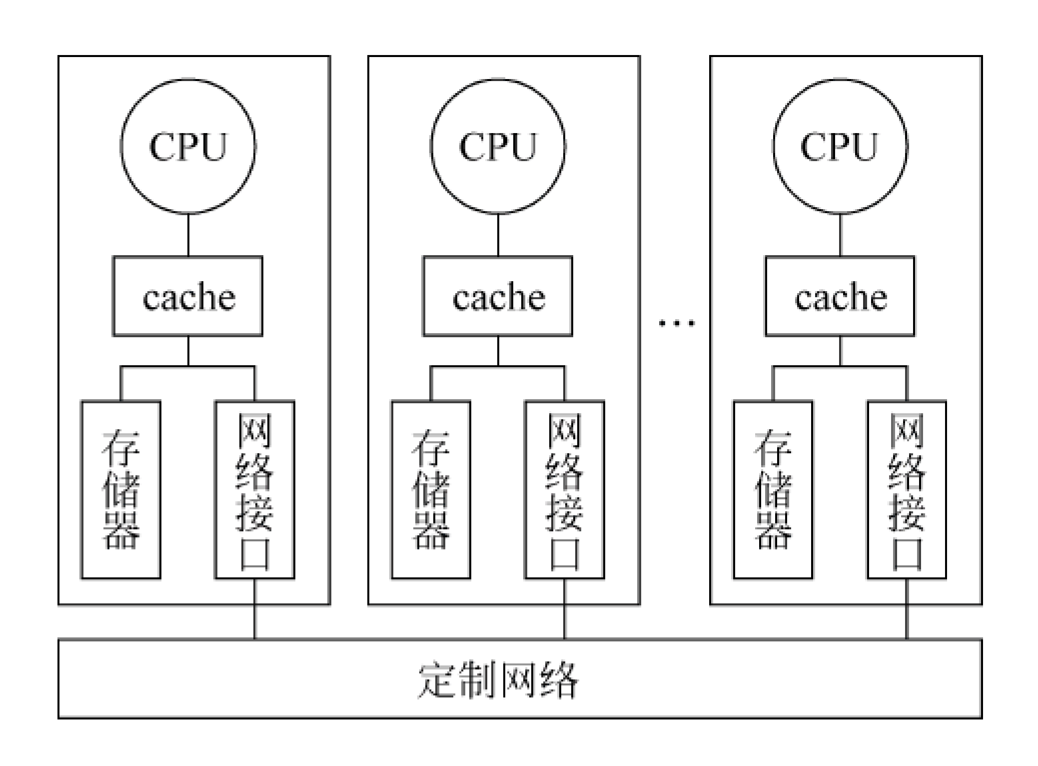


图 2-7 MPP 结构图

针对 MPP 系统的研究虽已有较长的历史,但由于研制费用高,故主要由大公司或研究机构研制生产。尤其是超大规模 MPP 系统(如峰值运算速度在每秒一万亿次以上浮点运算的系统)的研制,通常为政府行为,如美国的 ASCI 计划和 CIC (Computing Information and Communication Program)计划中的高端并行机。ASCI 计划由美国能源部出资,在美国三大军用实验室,使用由 IBM、Intel 与 SGI 三家公司研制的超级计算机进行核武器测试。MPP 系统过去主要用于科学计算、工程模拟等以计算为主的场合。目

前，MPP 也广泛应用于商业和网络应用中，例如应用于数据仓库、决策支持系统和数字图书馆等中。

2. 特点

MPP 通常具有如下特点：

- 在处理结点中使用商品化处理器，且每个结点有一个或多个处理器；
- 在处理结点内使用物理上分布的存储器；
- 使用具有高通信带宽和低延迟的互联网，结点之间彼此是紧耦合的；
- 能扩展到成百上千个处理器；
- 它是一个异步多指令流多数据流(MIMD)计算机，通常采用锁方式进行消息传递操作来实现同步，也有采用共享变量来实现同步操作的实例；
- 其上的程序由多个进程组成，每个进程拥有自己的私有地址空间，通过显式的消息传递实现进程间通信，数据分布对于用户来说不是透明的。

3. 关键技术

MPP 的特殊之处在于系统被设计成可扩展至数千个处理器，且主存、I/O 能力和带宽能成比例地增加。为提高其可扩展性，MPP 采用了如下技术：

- 使用物理上表现为分布式的主存体系结构，它提供比集中式主存体系结构更高的总主存带宽，因此具有潜在更高的可扩展性；
- 处理能力与主存和 I/O 能力之间的平衡性，若没有成比例的高速主存和 I/O 子系统，那么数据不可能以足够快的速度被送入处理器，高速处理器就将几乎毫无价值；
- 计算能力与并行性和交互能力之间的平衡性，减小进程/线程管理、通信以及同步的开销。

MPP 系统必须接受严酷的可用性考验。据统计，一个常规的 MPP 系统，每 1000 个处理器就会有 1 个处于失效状态。因此，还需要采用如下技术来保证 MPP 的可用性：

- 具有隔离的冗余设备，当某个主组件失效时，其辅助组件承担其提供的服务，且主组件和辅助组件需要隔离开，以避免其同时失效；
- 能够实现故障接管，即当一组件发生故障时，通过故障诊断、故障通知、故障恢复，能够使系统剩余部分承担故障组件的工作，从而实现任务迁移。

此外，MPP 系统采用虚拟化单一系统映像技术，即在不同层次上实现统一的系统映像，使用户可以将整个系统视为一个整体，从而简化系统管理，降低操作难度。

MPP 系统与机群系统的关键差别在于结点之间的通信。在机群系统中，结点之间通常由标准局域网相连；而在 MPP 系统中，结点之间由高带宽、低时延的高速专用网络互

联,同时还提供专用通信软件以实现高性能。

这里需要特别指出的是,随着标准网络技术的飞速发展,MPP 系统正逐渐被机群系统所取代,并且随着并行计算机技术的发展,MPP 系统与其他类型的并行计算机之间的界限变得越来模糊。

4. 典型实例

下面介绍一个 MPP 的典型实例:Cray 公司的 T3E。

于 1995 年交付使用的 Cray T3E 是 1993 年生产的 Cray T3D 系统的后继产品。它使用了更快的部件,并在体系结构方面做了一些修改以提高性能。

Cray T3E 是一个分布共享存储 NCC-NUMA 的多处理机。该系统由多个处理单元 PE(Processing Element)组成,PE 之间由一个三维双向环网进行互连以提供快速的通信,并由千兆环通道提供与 I/O 设备的连接。

Cray T3E 的每个 PE 中有一个 DEC Alpha 21164 微处理器,其外部是一个 shell 电路,包括一个本地主存、一个控制芯片和一个路由芯片。该系统的峰值速度可达 600MFLOPS。

本地主存提供 64MB～2GB 的容量以及 1.2GB/s 的峰值带宽。路由芯片有 7 个双向端口,1 个连接到 PE,其余 6 个连接到三维环网的 6 个链接上。

定制的控制芯片实现分布共享存储,它由所有 PE 中的本地主存组成。每个处理器可以访问任意 PE 中的主存,每个 PE 可以通过千兆环通道访问任意 I/O 设备。该控制芯片同时负责支持时延隐藏和有效同步。

Cray T3E 的处理单元没有主板级高速缓存,而是使用 DEC Alpha 21164 微处理器中的高速缓存。片内的高速缓存有两级:第一级由一个 8KB 指令高速缓存和一个 8KB 数据高速缓存组成;第二级是一个三路组相连(Three-Way Set-Associative)的 96KB 统一高速缓存,用于存储指令和数据。Cray T3E 之所以不使用主板级高速缓存是为了提高主存储器的带宽。

Cray T3E 的结构如图 2-8 所示。

Cray T3E 是一个自主系统,它运行 Cray 64 位 UNIX 系统(UNICOS)的一个变体,称之为 UNICOS/mk。它是一个分布式的操作系统,在其核心层提供单一系统映像。Cray T3E 提供一个集成环境,支持共享变量、消息传递和数据并行编程。

UNICOS/mk 操作系统分为本地和全局服务器。Cray T3E 的 PE 分为用户 PE 和系统 PE。其中,用户 PE 运行用户的应用和命令,系统 PE 负责提供全局操作系统服务。每个用户 PE 包括本地服务器和一个使用 Chorus 技术的 UNIX 微内核。

所有特定的进程请求都由本地服务器和 UNIX 微内核进行处理,包括主存分配和消息/数据传递。全局服务器提供系统范围的服务,包括进程管理、文件空间分配、调度、安

全性和 I/O 管理等。

UNICOS/mk 操作系统使用作业自动恢复来支持可用性，其方法有由 UNIX 微内核支持的检测点/重启、共享文件系统等。可用性包括提供资源管理、系统管理、系统监控、作业调度和安全性服务等一系列的工具。

为获得可扩展的 I/O，UNICOS/mk 操作系统实现了分布式文件系统管理。用户 PE 中的本地文件服务器提供无缓冲的读写请求服务，只有不太常用的请求（如文件打开和关闭）才需要使用全局文件服务器。多文件服务器能够实现并行 I/O。

Cray T3E 提供了支持 Fortran 90、C 和 C++的优化编译器、一系列优化和并行化的科学与数学库。Cray T3E 支持使用 Fortran 90 和 HPF 语言的数据并行编程模式，支持使用 PVM 和 MPI 库的消息传递编程模式，支持使用 Cray 共享存储库 SHMEM 和 CRAFT 编译器命令与库例程的共享变量编程模式。它们之间还可以混合使用。

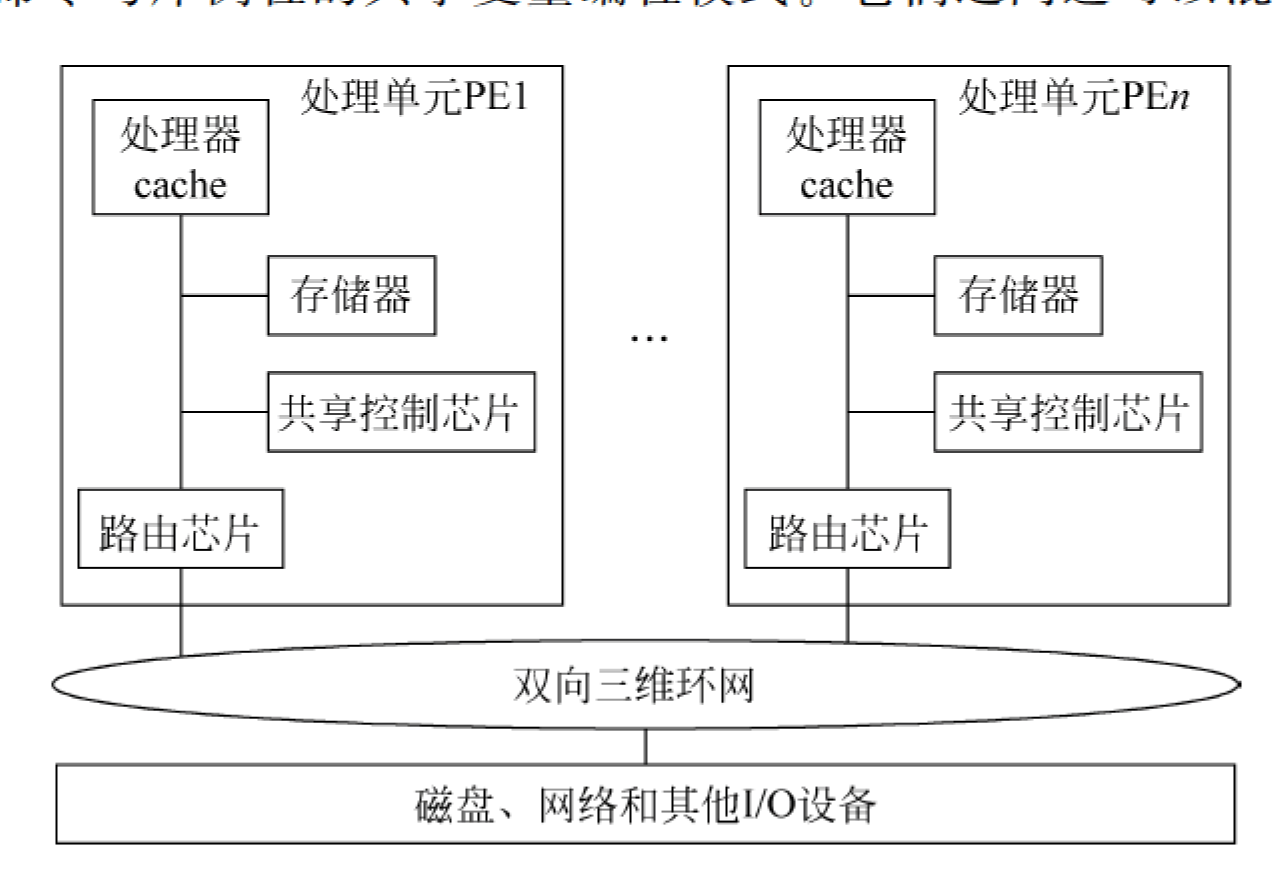


图 2-8　T3E 结构图

2.1.6　机群系统

1. 简介

机群系统（Cluster）是相互连接的多个同构或异构的独立计算机的集合体，结点之间通过高性能互联网连接。每个结点都有自己的存储器、I/O 设备和操作系统，可以作为单机使用。结点之间相互协同工作来完成较复杂的并行性任务，从而成为一个多处理机系统。在最近的 10 年里，机群系统以其高性价比、高可扩展性和结构的灵活性逐渐在越来越多的领域得到应用，成为高性能计算机家族中发展最快的一员。

机群系统的结构（如图 2-9 所示）十分灵活，系统中的各个结点可以是完全不同的结构，结点之间采用商用网络进行互联。

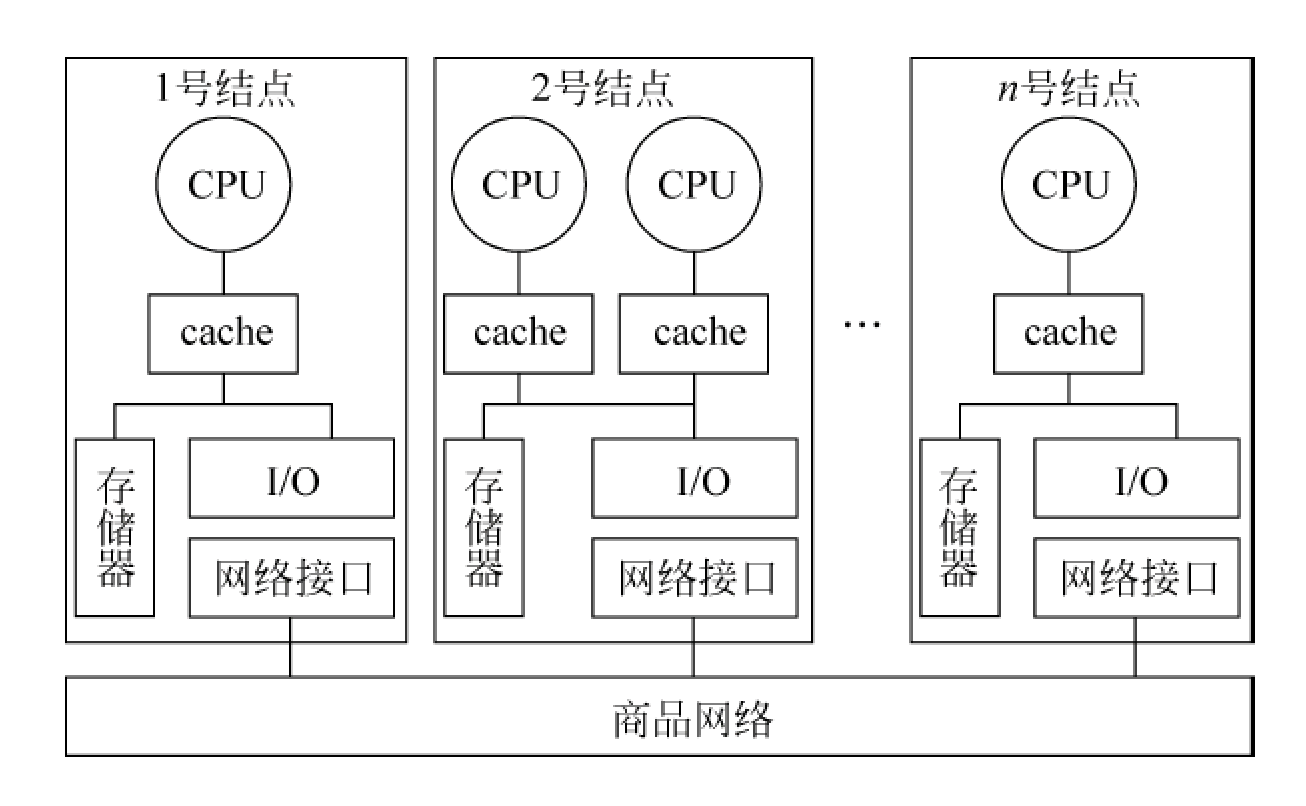


图 2-9　机群系统结构图

2. 特点

机群系统具有如下特点：

- 灵活性强，机群系统的各结点构成十分灵活，每个结点可以是单处理机，也可以是SMP或其他类型的并行计算机。
- 结点独立性，每个结点都是一个完整的系统，有自己的本地磁盘和操作系统。
- 可靠性高，机群系统中每个结点都是独立的PC或工作站。某个结点的失效不会影响其他结点的正常工作，能够完成故障接管和任务迁移，保证系统的可靠性。
- 可扩展性强，由于机群系统结构的灵活性和松耦合方式，机群系统的硬件容易被扩充和替换，可根据需求增加结点的数量。
- 系统开发周期短，机群系统大多采用商品化的PC和工作站作为结点，并通过商用网络连接在一起，编程方法成熟且有继承性，便于开发者快速上手，无须适应新的环境，能够大大节省研究时间。
- 性价比高，由于传统并行计算机大多属于定制机，生产数量较小，所以价格昂贵。而机群系统的结点和网络采用的是大批量生产的计算机产品，成本相对较低，具有更高的性价比。

3. 关键技术

机群系统具有很多其他并行计算机不可比拟的优势，同时也面临许多设计问题，比如可用性、良好的性能、可扩展性等。具体来说，涉及到如下几个方面的问题。

1）高效通信

机群系统比其他并行计算机更需要一个高效的通信子系统，因为机群系统有以下特点：

- 结点复杂度高，不可能做到紧耦合；
- 结点之间的连接线路比较长，带来了较大的通信延迟，同时也带来了可靠性、串道等问题；
- 机群系统一般使用标准通信协议下的商用网络，通信协议的开销比较大。

因此，高效的通信子系统是提高机群系统性能的关键，对机群系统的并行加速比、并行效率、可扩展性以及系统的适用范围等有着十分重要的影响。加快通信速度、减少通信开销、使系统的资源主要用于计算等是机群系统研制的重要内容。

提高机群系统的通信性能需要考虑如下几个方面：

- 采用新型高速网络，提高网络带宽，目前已有 1Gb/s 的高速商用网络；
- 设计新的通信协议，降低通信延迟，尽量减少网络协议对主机操作系统的服务请求，最大限度地实现通信与计算的重叠。

2）负载均衡和任务调度

在机群系统中，一个大的任务往往由多个子任务组成。分配到各个处理结点上并行执行的子任务，被称为负载。当整个系统的任务较多时，分配给各个处理结点的负载可能并不均衡，此时整个系统的利用率就会降低。因此，机群系统必须做好任务调度以尽可能达到负载均衡。

一般来说，负载均衡技术有静态和动态两类方法。静态负载均衡方法就是在编译时针对用户程序中的各种信息和机群系统的特点对用户的任务进行静态划分。静态负载均衡方法需要建立在对任务的总体掌握和对机群系统进行深入分析的基础上，因此常作为一种理论方法。动态负载均衡方法是通过分析机群系统的实时负载信息，动态地将任务在各处理结点之间进行分配和调整，以消除系统中负载分布的不均匀性。动态负载均衡的特点是算法简单，实时调度，但同时也增加了系统的额外开销。

3）单一系统映像

单一系统映像（Single System Image，SSI）是机群系统的一个重要特征，它可使机群系统在使用、控制、管理和维护上更像一台工作站。单一系统映像可以让机群系统灵活地采用集中式或分布式的管理和控制，大大简化系统的管理，降低操作员错误带来的风险。

要实现单一系统映像，需要分别实现以下几个目标：

- 单一系统入口；
- 单一文件层次目录结构；
- 单一输入输出；
- 单一进程空间。

4）容错性与可用性

机群系统的结点众多，结构复杂，系统中结点发生故障的概率较高。因此，容错性和可用性是一个机群系统必须解决的问题。一般来说，要求机群系统能够自动恢复瞬时或

间歇性故障,能够人工恢复永久故障,支持在线维修和处理机资源的排他或限时使用,此外为了实现动态负载均衡,还应能进行进程迁移。

目前,常采用检查点设置和回卷恢复(Checkpoint and Rollback Recovery)技术来实现上述目标。在程序运行过程中设置检查点,保存进程状态中那些决定程序正确执行的关键内容。当系统出现故障时,程序回卷到最近的检查点继续执行,无须从头开始。回卷的各目标检查点所保存的进程状态与当时的通信状态组成一致性全局状态,并采用同步回卷技术以避免活锁。

4. 典型实例

下面以IBM公司的SP2机群系统为例进行说明。

SP2系统早先划归为MPP,但由于其采用了机群技术,所以从广义上讲,它也是机群系统的一个典型实例。IBM公司在1991年秋天涉足MPP的商业化,启动了SP(Scalable POWER parallel)项目。在1992年2月成立了一个开发小组,在1993年4月发布了它的第一个产品SP1,继而在1994年7月发布SP2系统。到1998年,全球已安装超过3000台SP系统。其中,在1997年的人机大战中战胜世界国际象棋冠军卡斯帕罗夫的“深蓝(Deep Blue)”就是一台采用了30个RS/6000工作站的IBM SP2机群系统。

一个SP2系统(如图2-10所示)可包括2～512个结点,每个结点都有自己的局部存储器和局部磁盘,所有的结点均连接两个网络:一个普通的以太网和一个高性能开关(High Performance Switch,HPS)。以太网的速度比较慢,但可在HPS失效时作为备用连接使用。当HPS和相关软件正在开发和改进时,可利用以太网对系统的其他部分进行开发、调试、测试以及使用,此外以太网还可用于系统监视、启动、载入、测试和管理等。

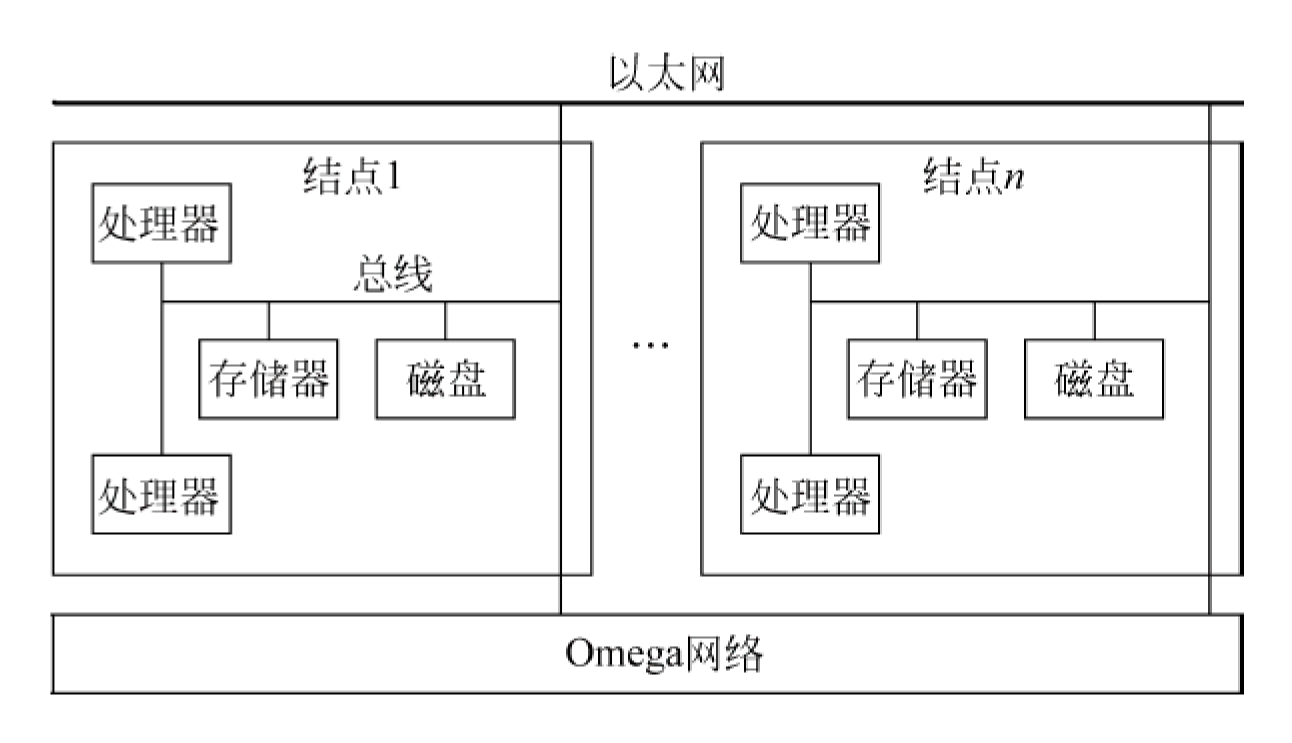


图2-10　SP2机群结构图

HPS是一个4级Omega网络(一种多级互联网,又称多级洗牌交换网络),由40MHz时钟驱动,由8个开关帧和4块开关板构成。它采用虫蚀寻径方法,理论上一个8位的

数据片在无竞争时通过 HPS 只需 20 个时钟周期(即 500ns)。因此,HPS 在无竞争时的理论延迟是很小的,但实际延迟比该值要高得多。比如,一个进程发送一个空包给另一个进程至少需要 40μs,这种消息传递的延迟大部分是由软件开销造成的。SP2 的结构如图 2-10 所示。

SP2 提供如下三种物理结点类型以有效支持配置的灵活性:宽结点(Wide Node)、窄结点(Thin Node)和窄结点 2。这三种类型的结点的差别主要在于存储器容量、数据位宽和 I/O 总线槽数的不同,但是所有的结点都使用了时钟频率为 66.7MHz 的 POWER2 微处理器。每个处理器都有一个 32KB 的指令高速缓存、256KB 的数据高速缓存、一个指令和分支控制部件、两个整数部件、两个可在一个时钟周期内执行乘法和加法的浮点部件。

2.1.7 并行计算机传统体系结构的比较与分析

SMP、DSM、MPP 和机群系统作为四种最经典的并行计算机模型,基本包括了从简单到复杂、不同时期、不同用途的并行计算机。学习这些经典并行计算机的体系结构和设计方法,了解它们的异同和各自特点,对于开发更先进的并行计算机具有重要的意义。四种典型并行计算机特征的比较如表 2-2 所示。

同时也应看到,并行计算机的设计具有很强的灵活性,上述四种类型的并行计算机不具有排他性的分别。一些经典的并行计算机就是吸收了多种不同类型计算机的特点而开发出来的。如 Cray 公司的 T3D、T3E 系列机型就是采用 DSM 结构的 MPP,IBM 公司的 SP 系列机型同时具备机群和 MPP 的特点。

表 2-2　四种典型并行计算机特征的比较

属性	SMP	DSM	MPP	Cluster
互联网	总线/交叉开关	定制网络	定制网络	商用网络/以太网
地址空间	单地址空间	单地址空间	单/多地址空间	多地址空间
存储器	集中存储	分布式存储	分布式存储	分布式存储
通信机制	共享存储	共享存储	消息传递/共享存储	消息传递
代表机型	Sun T1	Stanford DASH	Intel Paragon	Berkeley NOW
	SGI Challenge	Origin 2000	Cray T3E	IBM SP2



一般来说,SMP 结构简单,采用共享存储,扩展性差,适用于对扩展性要求不高的小规模计算;DSM 吸收了共享存储与分布存储的优势,编程简单同时兼顾可扩展性,是现在设计并行计算机的一种典型方法;MPP 作为曾经的高性能计算界的主流机型,拥有良好的可扩展性,但其紧耦合的特点影响了灵活性,正在逐渐退出历史舞台;机群系统以其

良好的可靠性、灵活性和可扩展性，成为高性能计算机的主流机型，在最新的全球超级计算机 top500 排行榜中，机群系统已经占到 85%以上。这四种并行计算机各具特点，适用于不同环境和任务。只有根据需求合理的使用它们，才能发挥并行计算机应有的性能优势。

2.2　多核 CPU

中央处理器(Central Processing Unit,CPU)，又称为微处理器，是现代计算机的主要部件之一。其功能主要是解释计算机指令以及处理计算机软件中的数据，所谓的可编程性主要是指对 CPU 的编程。其内部结构大概可以分为控制单元、算术逻辑单元和存储单元等几部分，这几部分相互协调，对命令和指令进行分析、运算，并控制计算机各部分协调工作。

CPU 从最初发展至今已经有三十多年的历史，由于 CPU 具有体积小、重量轻、功耗低、功能性强、结构灵活、价格低廉等特点和优点，因此得到了广泛的应用，也使得计算机深入到人类社会生产和生活的各个方面。目前，计算机已经成为人们工作和生活中不可缺少的工具，人类社会已经进入信息时代。

2.2.1　处理器架构

1. 计算机与处理器

CPU 是一台计算机的运算核心和控制核心，计算机中所有操作都由 CPU 负责读取指令，对指令译码并执行指令。按照处理信息的字长，可将 CPU 分为 8 位处理器、16 位处理器、32 位处理器以及 64 位处理器等；按照内部核心的数目，又可将 CPU 分为单核处理器、多核处理器和众核处理器等。

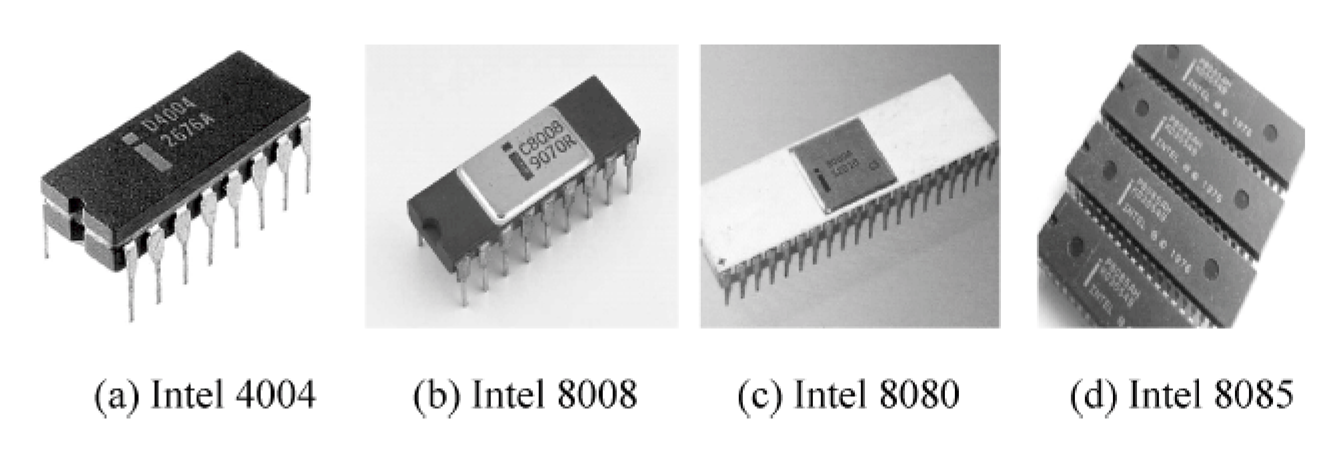
2. 单核处理器概述及其发展历史

1971 年，Intel 公司推出了 4 位处理器 4004，如图 2-11(a)所示。片内集成了 2250 个晶体管，晶体管之间的距离是 10μm，能够处理 4 位的数据，每秒运算 6 万次，运行的时钟频率为 108kHz，有 ROM、RAM 以及 I/O 的接口。该款芯片是第一款真正意义上的 CPU。但是由于性能很差，其市场反应不是很理想。

1972 年，Intel 公司推出了世界上第一款 8 位处理器 8008，如图 2-11(b)所示。8008 是 4004 的 8 位版本，8008 可以支持最大 16KB 的内存。

1974 年，Intel 公司推出 8 位处理器 8080，如图 2-11(c)所示。8080 的时钟频率为 2MHz，集成了 6000 只晶体管，每秒运算 29 万次，具有 16 位地址总线和 8 位数据总线，包

含 7 个 8 位寄存器,支持 16 位内存,同时还包含一些输入输出端口,有效解决了外部设备的内存寻址能力不足的问题。之后 Intel 公司又推出的 8 位处理器 8085,如图 2-11(d)所示。Intel 8085 可以向前兼容 Intel 8080。



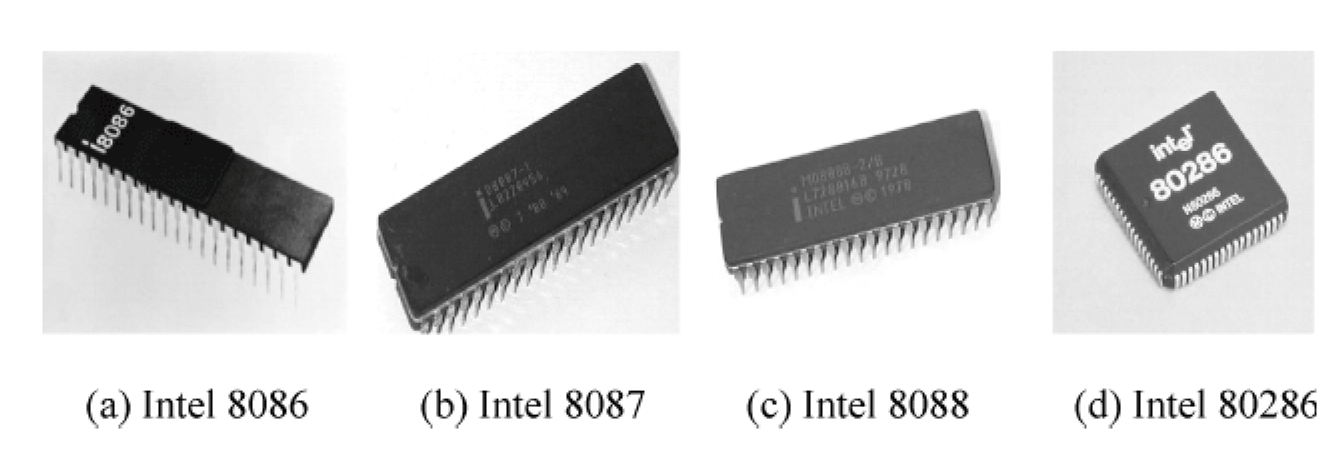
(a) Intel 4004 (b) Intel 8008 (c) Intel 8080 (d) Intel 8085

图 2-11 Intel 公司推出的 4 位和 8 位处理器

1978 年,Intel 公司推出了其 16 位处理器的典型代表 8086 以及数字协处理器 8087,如图 2-12(a)和(b)所示。8086 处理器的最高主频为 8MHz,具有 16 位数据通道,内存寻址能力为 1MB。8086 与 8087 使用相互兼容的指令集,但在 8087 的指令集中增加了一些专门用于处理对数、指数和三角函数等数学计算的指令,人们将这些指令集统称为 x86 指令集。Intel 把基于 32 位 x86 指令系统的个人计算机称为英特尔体系 32(Intel Architecture-32,IA-32)。虽然之后 Intel 又陆续生产出第二代、第三代等更先进和更快的 CPU,但都仍然兼容原来的 x86 指令集。

1979 年,Intel 公司推出了 16 位处理器 8088,如图 2-12(c)所示。8088 内含 2.9 万个晶体管,时钟频率为 4.77MHz,具有 20 位地址总线,16 位内部数据总线,8 位外部数据总线,内存寻址能力为 1MB。1981 年,8088 被首次用于 IBM PC 中,PC 的第一代 CPU 便由此开始。

1982 年,Intel 公司推出了 16 位处理器 80286,如图 2-12(d)所示。80286 内部包含 13.4 万个晶体管,时钟频率达到了 20MHz。其内外部数据总线均为 16 位,地址总线为 24 位,内存寻址能力为 16MB,可使用实模式和保护模式两种工作方式。

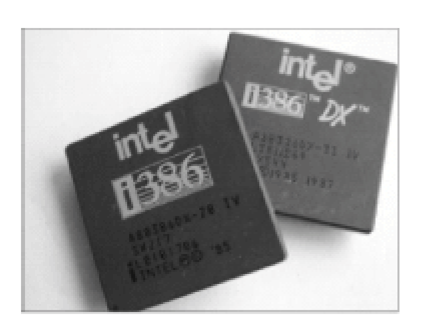
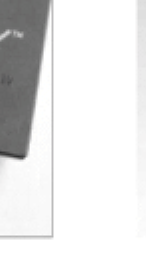


(a) Intel 8086 (b) Intel 8087 (c) Intel 8088 (d) Intel 80286

图 2-12 Intel 公司推出的 16 位处理器

1985 年,Intel 公司推出了 x86 架构中第一款 32 位处理器 80386,如图 2-13(a)所示。80386 内部包含 27.5 万个晶体管,刚推出时的时钟频率为 12.5MHz,之后逐步提高到

33MHz。具有 32 位的内部数据总线、外部数据总线和地址总线，内存寻址能力为 4GB。80386 除了具有实模式和保护模式之外，还增加了一种虚拟 86 的工作方式，可以通过同时模拟多个 8086 处理器来提供多任务处理能力。80386 处理器没有内置协处理器，不能执行浮点运算指令。如果需要进行浮点运算，必须额外购买昂贵的 80387 协处理器芯片，如图 2-13(b)所示。Intel 公司推出 80386 处理器之后，AMD 公司推出了相应的 32 位处理器 AMD Am386DXL-40，如图 2-15(a)所示。

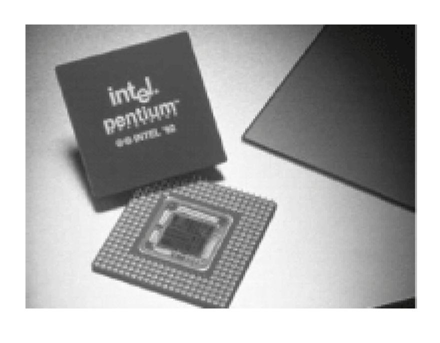
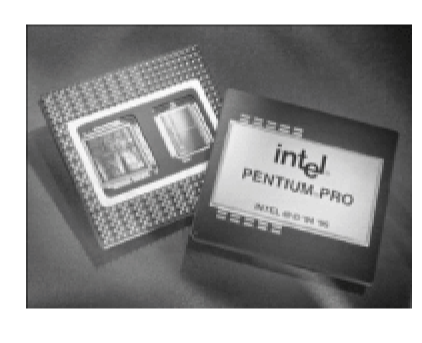
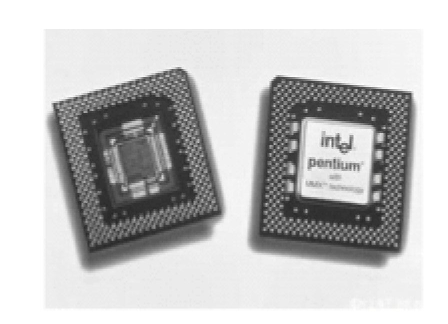

(a) Intel 80386　(b) Intel 80387　(c) Intel 80486

图 2-13　Intel 公司推出的 32 位处理器

1989 年，Intel 公司推出了 32 位处理器 80486，如图 2-13(c)所示。80486 内部包含了 125 万个晶体管，时钟频率由 25MHz 逐步提升到 100MHz。80486 是 Intel 公司第一款内部包含数字协处理器的 CPU，并在 x86 系列中首次使用了 RISC(精简指令集)技术，从而提高了每时钟周期执行指令的速度。80486 还采用了突发总线方式，大大提高了处理器与内存的数据交换速度。Intel 公司推出 80486 处理器之后，AMD 公司推出了相应的 32 位处理器 AMD Am486dx2-80，如图 2-15(b)所示。

1993 年，Intel 公司推出了新一代 x86 架构 32 位处理器 Pentium，如图 2-14(a)所示。Pentium 内部集成了 310 万个晶体管，时钟频率由 60MHz 逐步提升到 200MHz 以上。AMD 公司推出了 AMD K5 处理器(如图 2-15(c)所示)来应对 Intel Pentium 处理器，但是由于 Pentium 的性能更佳，Intel 逐渐占据了处理器的大部分市场。

1996 年，Intel 公司推出了基于 x86 架构的 32 位处理器 Pentium Pro，如图 2-14(b)所示。Pentium Pro 的内部集成了 550 万个晶体管，时钟频率为 133MHz，处理速度几乎是 100MHz 的 Pentium 的 2 倍。Pentium Pro 的 L1(片内)cache 为 8KB 指令和 8KB 数据。Pentium Pro 的一个封装中除 Pentium Pro 芯片外还包括一个 256KB 的 L2 cache 芯

(a) Intel Pentium　(b) Intel Pentium Pro　(c) Intel Pentium MMX

图 2-14　Intel 公司推出的 Pentium 系列处理器

片，两个芯片之间用高频宽的内部通信总线互连，处理器与高速缓存的连接线路也被安置在该封装中，这样就使得高速缓存能更容易地运行在更高的频率上。同时，Pentium Pro具有一项称为“动态执行”的创新技术。

1997年，Intel公司推出了Pentium系列的改进版本，即Pentium MMX，如图2-14(c)所示。Pentium MMX在原Pentium的基础上进行了重大的改进，增加了片内16KB数据缓存和16KB指令缓存，4路写缓存以及从Pentium Pro、Cyrix继承而来的分支预测单元和返回堆栈技术，特别是新增加的57条MMX多媒体指令。

同年，Intel公司推出了基于x86架构的处理器Pentium Ⅱ，如图2-16(a)所示。Pentium Ⅱ基于Pentium Pro架构，采用0.35μm的制造工艺，内部集成了750万个晶体管，加入MMX指令集，集合了32KB片内L1 cache和8个64位的MMX寄存器，L2 cache是具有512KB四路级联片外同步突发式SRAM高速缓存。采用了双独立总线结构，其中一条总线连接二级高速缓存，另一条连接到内存。

同时，AMD公司于1997年也推出了采用0.35μm制造工艺的处理器K6，如图2-15(d)所示。K6内部集成了880万个晶体管，拥有32KB数据L1 cache和32KB指令L1 cache。继K6处理器之后，AMD公司于1998年又推出了0.25μm工艺制造的处理器K6-2，如图2-15(e)所示。它拥有930万个晶体管，有64KB L1 cache(32KB指令集和32KB数据)。

1999年，AMD公司推出了K7处理器，并将其正式命名为Athlon，如图2-15(f)所示。首款Athlon代号为Thunderbird(暴风鸟)。K7最早采用的是0.25μm的制造技术，而后采用0.18μm铜互连技术。K7采用200MHz的外频，内建512KB共享L2 cache。K7加强了整数、浮点运算和多媒体运算的能力。



(a) AMD Am386DXL-40



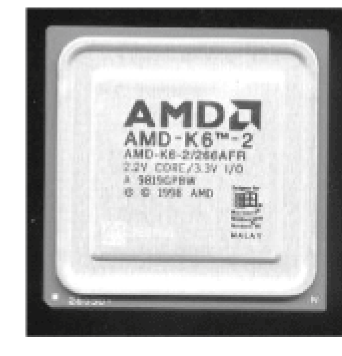
(b) AMD Am486dx2-80



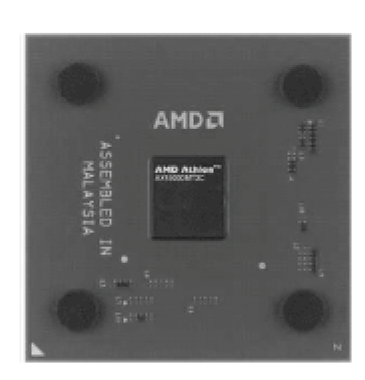
(c) AMD K5



(d) AMD K6



(e) AMD K6-2



(f) AMD K7

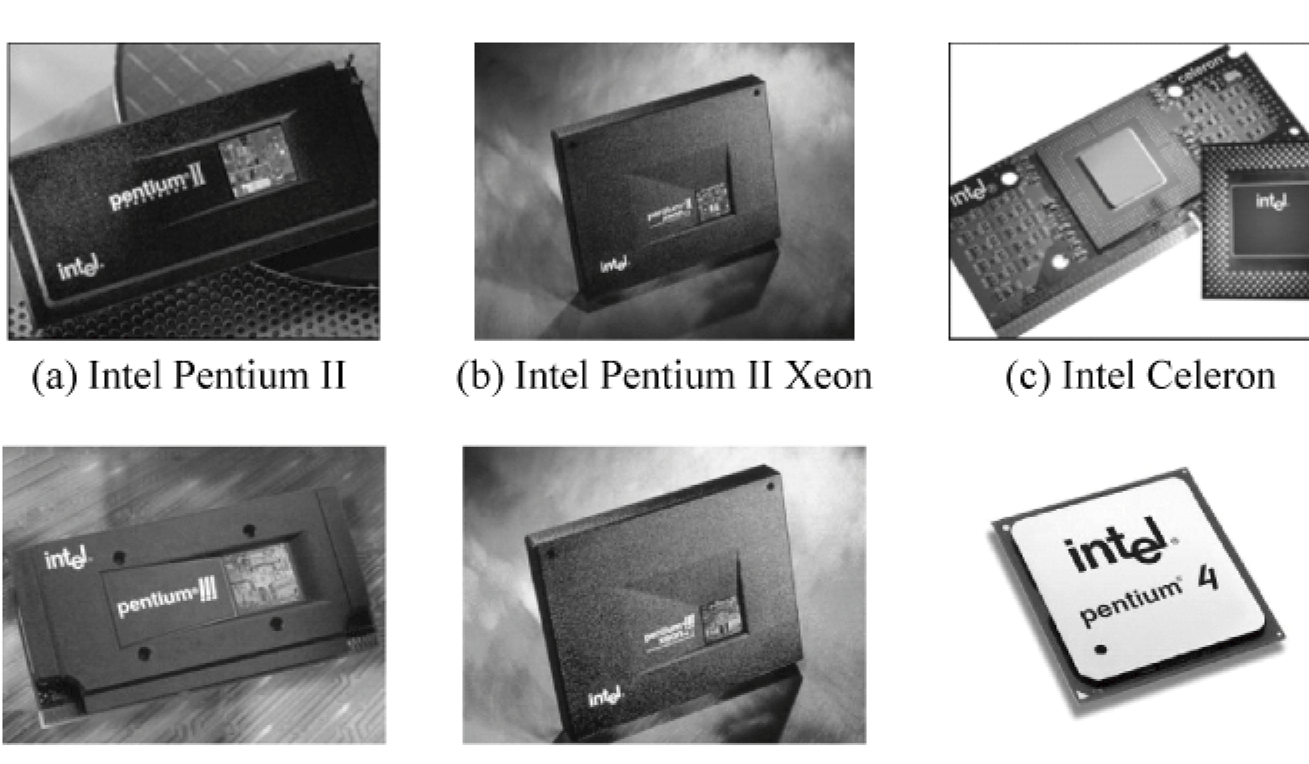
图2-15 AMD公司推出的32位处理器

1998—1999 年间，Intel 公司面向服务器和工作站市场，推出了比 Pentium Ⅱ功能更加强大的 Pentium Ⅱ Xeon 处理器，如图 2-16(b)所示。Pentium Ⅱ Xeon 基于 Pentium Ⅱ核心架构，具有 512KB 或 1MB、400MHz 的高速缓冲存储器，在处理器、RAM 和 I/O 器件之间传递数据的高速总线，能提供 36 位地址的扩展服务器内存结构。Pentium Ⅱ Xeon 不但有更快的速度与更大的缓存，更重要的是可以支持多达 4 路或者 8 路的 SMP 对称多 CPU 处理功能(必须配合专门的服务器主板才能使用)。

1998 年，Intel 公司面向低端市场推出了一款廉价的处理器 Celeron，如图 2-16(c)所示。Celeron 起始时钟频率是 266MHz，开始没有 L2 cache，后来因整数性能太差的原因加入了 128KB 或 256KB 的 L2 cache。用于移动处理的 Celeron-M 处理器则具有 1MB 的 L2 cache，凭借其良好的超频性能和便宜的价格，在当时赢得了许多用户及超频玩家的喜爱。

1999 年，Intel 公司发布了 Pentium Ⅲ处理器，如图 2-16(d)所示。从 Pentium Ⅲ开始，英特尔又引入了 70 条新指令(SIMD 与 SSE)，主要用于因特网的流媒体扩展(提升网络演示多媒体流、图像的性能)、3D、流式音频、视频和语音识别功能的提升。Pentium Ⅲ有如下三种核心：

- 最初的版本是 Katmai，采用 0.25μm 制造工艺，加入了 SSE，改进 L1 cache 控制器，使用半速 512KB L2 cache，采用 Slot 1 接口；
- 最流行的版本是 Coppermine(铜矿)，于 1999 年推出，采用 0.18μm 制造工艺，采用低延迟全速 256KB L2 cache，采用了新的 Socket 370(FC-PGA)和 Slot 1 两种接口；
- 2001 年推出的 Tualatin(图拉丁)，采用 0.13μm 制造工艺，采用低延迟全速 256KB L2 cache，采用了新的 Socket 370(FC-PGA)接口。



(a) Intel Pentium II　(b) Intel Pentium II Xeon　(c) Intel Celeron
(d) Intel Pentium III　(e) Intel Pentium III Xeon　(f) Intel Pentium 4

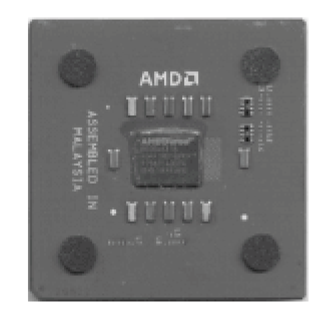
图 2-16　Intel 公司推出的 Pentium Ⅱ至 Pentium 4 处理器

1999 年,Intel 公司还发布了 Pentium Ⅲ Xeon 处理器,如图 2-16(e)所示。Pentium Ⅲ Xeon 具有 2MB L2 cache,更大的缓存有助于提高性能,也继承了 Pentium Ⅲ新增的 70 条指令集,以更好地执行多媒体、流媒体应用软件。除了面对企业级市场以外,Pentium Ⅲ Xeon 加强了电子商务应用与高级商务计算的能力。在缓存速度与系统总线结构上,也有很大的进步,大幅提升了性能,并为更好的多处理器协同工作进行了优化设计。

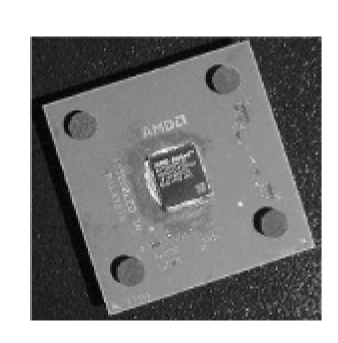
2000 年,Intel 公司发布了 Pentium 4 处理器,如图 2-16(f)所示。采用 Intel NetBurst 技术的 Pentium 4,内部集成了 4200 万个晶体管,到了改进版的 Pentium 4 (Northwood)更是集成了 5500 万个晶体管,并且开始采用 0.18μm 制造工艺,初始时钟频率就达到了 1.5GHz。其技术特点是采用了超级通道技术,可使 Pentium 4 在 20 级通道里执行软件指令(Pentium Ⅲ只有 10 级通道),首次推出的 400MHz 系统总线可加速数据在处理器和主存储器之间的传输。此外,Pentium 4 增加了 144 条新指令以加速处理视频、音频和 3D 的应用。Pentium 4 也有低端 Celeron(通常称为 Celeron 4)和用于 SMP 配置的高端 Pentium 4 Xeon 版本。2002 年,Intel 公司又推出了内含创新的超线程技术(hyper-threading,HT)的新一代 Pentium 4 处理器。超线程技术就是一个 CPU 同时执行多个程序而共同分享一个 CPU 内的资源,理论上像两个 CPU 一样在同一时间执行两个线程。为实现此目的,Pentium 4 需要多加入一个 Logical CPU Pointer(逻辑处理单元),而其余部分如 ALU(整数运算单元)、FPU(浮点运算单元)、L2 cache 则保持不变。

2000 年,AMD 公司为了争夺处理器的低端市场份额,在简化 Athlon 处理器的基础上推出了 Duron 处理器,如图 2-17(a)所示。AMD 公司前后共发布了基于 Spitfire、Morgan、Applebred 共计三种核心的 Duron 处理器。Duron 内部集成了 2500 万个晶体管,具有 200MHz 外频,内置 128KB 的 L1 cache 和 64KB 的全速 L2 cache,工作电压为 1.5V。这些特点均符合 AMD 面对低端市场的策略,即低成本、低功耗但高性能。在浮点性能上,基于 K7 体系 Duron 明显优于采用 P6 核心设计的 Intel 系列处理器。

2001 年,AMD 公司推出了 Athlon XP(eXtreme Performance)处理器,如图 2-17(b)所示。Athlon XP 是在 Thunderbird 核心的 Athlon 基础上改进而来,新增了对 SSE 指令



(a) AMD Duron



(b) AMD Athlon XP



(c) AMD Sempron

图 2-17 AMD 公司推出的 32 位处理器

集的支持，可以支持 3DNow! 指令。AMD 公司前后共发布了包括 Palomino、Thoroughbred、Barton、Thorton 在内的四种核心的 Athlon XP 处理器，它们全部采用 Socket A 接口，采用更为先进的 OPGA 封装技术。

2003 年，AMD 公司推出了 Barton 处理器。Barton 采用新款 Athlon XP 核心，L2 cache 容量增加到原来的两倍，而其他设计基本没有变化。这样在不提高频率的情况下，Athlon XP 的性能得到了进一步的提高。

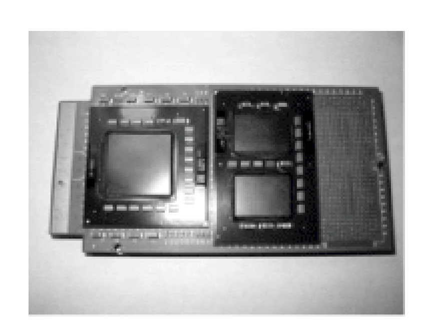
2004 年，AMD 公司推出了 Sempron 处理器，取代了之前的 Duron 系列，如图 2-17(c) 所示。与 Duron 一样，Sempron 也简化了 L2 cache。

在过去 20 年间，32 位的 x86 架构以无可比拟的性价比优势成为计算平台的标准。但是由于 x86 仍然基于 32 位技术，对于高端的企业级服务器与工作站应用显得力不从心。伴随着企业级计算应用的发展，64 位应用将越来越广泛，令 x86 向 64 位进行扩展势在必行，也成为统一 64 位计算标准的希望。

Intel 公司最初并不认为个人和移动领域需要 64 位的体系结构。为此，Intel 单独发布了专为 64 位市场而定制的 IA-64 架构以及相关的 64 位指令规格，即著名的显式并行指令计算(Explicitly Parallel Instruction Computing，EPIC)，并于 2001 年发布了名为 Itantium(安腾)的企业级 64 位处理器，如图 2-18(a)所示。Itantium 确实是高性能的处理器，但是其 IA-64 并不兼容 x86-32 指令集，导致大众应用被 Intel 划分到 64 位的范围之外。次年，Intel 又推出了 Itantium 2，仍然不兼容 x86-32 指令集。

2003 年，AMD 推出全球首款可与业内标准 x86 兼容的代号为 SledgeHammer 的 64 位处理器 AMD Opteron，如图 2-18(b)所示。AMD Opteron 基于 AMD K8 架构，兼容以往的 32 位指令集，也就是说，AMD Opteron 不但是一颗 64 位处理器，同时也是 32 位的。

同年，AMD 推出针对家用市场的 64 位处理器 Athlon 64 3000＋，如图 2-18(c)所示。Athlon 64 采用 Socket 754 的简化封装，拥有一个单通道内存控制器，可以与普通 DDR 内存模块搭配使用。Athlon 64 3000＋的时钟频率达到了 2.0GHz，L2 cache 为 512KB。


(a) Intel Itantium


(b) AMD Opteron


(c) AMD Athlon 64

图 2-18　早期的 64 位处理器

2.2.2　单核处理器发展瓶颈

从 Intel 公司于 1971 年推出的全球第一个通用型处理器 4004 开始，在一块芯片上集

成的晶体管数目越多,意味着运算速度(即主频)就更快。但是,面对不断涌现的计算机的新兴使用模式,用户对处理器的处理能力(即性能)及其年增幅提出了更高的要求。

提高处理器性能有两条途径:

- 提高处理器的主频;
- 提高每个时钟周期内执行的指令数(IPC)。

处理器架构的变化可以改变 IPC,效率更高的架构可以提高 IPC,从而提高处理器的性能。但是,通过改良同一代的架构来提高 IPC 的幅度却是非常有限的。所以,在单核处理器时代,通过提高处理器的主频来提高性能就成了唯一的手段。不幸的是,并非可以无止境地提高处理器的主频。如果通过提高主频来提高处理器的性能,就会使处理器的功耗以指数(三次方)而非线性(一次方)的速度急剧上升,很快就会触及所谓的“频率墙”。因此,功耗问题成了提高单核处理器性能的瓶颈。过快的能耗上升,迫使众多厂商转而寻找提高 IPC 的方法。

提高 IPC 可以通过提高指令执行的并行度来实现,而提高并行度有两种途径:

- 提高处理器架构的并行度;
- 采用多核架构。

其次,芯片上除了晶体管就是互连线,它的主要工作是把一个晶体管的处理结果传输给另一个晶体管。晶体管的集成度按摩尔定律变得越来越高,但是互连线的延迟并未随之变快。在这种情况下,互连线的延迟问题就成了提高单核处理器性能的瓶颈。克服互连线延迟增加的最好办法是使用一些较小的核组成一个多核的芯片,而不是以往的单核芯片。

随着晶体管数量的增加,芯片设计的空间、复杂度和验证难度都将大幅度增加。如果采用多个同一设计的处理器核,那么设计的复杂度就会大大减小,从而使得设计成本降低,出错的几率也相应减小。

为此,多核处理器的出现是摩尔定律与物理限制(功耗、互连线、设计复杂度)相互作用的结果。处理器上的晶体管数目越来越多,但却因功耗与互连线的限制而不能直接提供很高的性能,在一个处理器中集成多个简单的处理器核可以有效解决该问题。综上,多核处理器的出现是处理器发展到一定阶段的必然产物。

2.2.3 单芯片多处理器架构

1. 多核处理器概述

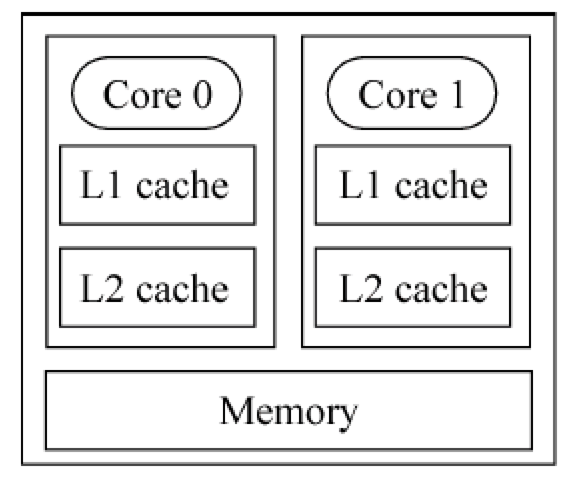
随着单个芯片上晶体管数目的增加,美国斯坦福大学提出了 Chip MultiProcessors(简称 CMP,也称为单芯片多处理器)。CMP 是指由单个芯片上的多个处理器核所构成的处理器系统,即多核处理器。其基本思想是:将大规模并行处理器中的 SMP(对称多

处理器)集成到同一芯片内,允许线程在多个处理器核上并行执行。通过在多个 CPU 核上分配工作负荷,同时依靠到内存和输入输出(I/O)的高速片上互连和高带宽管道,多核处理器的系统性能得以提升。较之以前的单核处理器,多核处理器能带来更高的性能和生产力优势,因而成为现在一种广泛普及的计算模式。

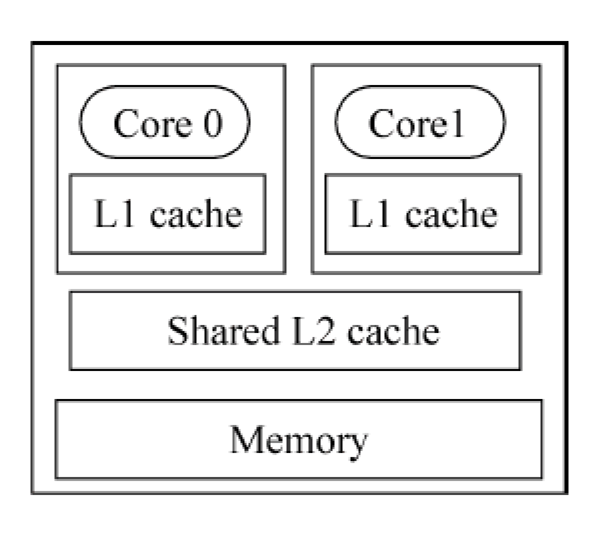
与使用线程级并行来提高多核处理器性能相类似的另一种方法就是同时多线程(Simultaneous Multithreading,SMT)技术。SMT 技术是一种体系结构模型,其目的是在现有硬件条件下,通过提高计算能力来提高处理器的性能。Intel 公司所实现的 SMT 技术就是超线程(Hyper-Threading,HT)技术。HT 技术实际上只有一个实际的物理处理器,但是从软件的角度看,存在多个逻辑处理器。它支持操作系统和应用程序将多个线程调度到多个逻辑处理器上,就像在多处理器系统上一样。虽然,采用超线程技术能同时执行两个线程,但它并不像两个真正的处理器(每个处理器都具有独立的资源)那样。当两个线程都同时需要某一个资源时,其中一个线程要暂时停止,直到另一个线程执行完毕之后该线程才能继续执行。因此,超线程的性能并不等同于两颗处理器的性能。

从体系结构的角度看,SMT 比 CMP 对处理器资源的利用率更高。但是随着超大规模集成电路工艺技术的发展,晶体管的尺寸不断缩小,这使得晶体管的门延迟不断减少,但互连线的延迟却不断变大。当芯片的尺寸减小到 0.18μm 甚至更小时,互连线的延迟已经超过晶体管的门延迟,成为限制电路性能提高的主要因素。在这种情况下,与 SMT 的集中式结构相比,CMP 的分布式结构由于全局信号较少,在克服互连线的延迟影响方面更具优势。同时,由于 CMP 结构是基于多个处理器内核设计的,每个核都比较简单,对优化设计非常有利,因此发展前景好。

CMP 可以在不同的存储层次上进行互连,据此可将 CMP 分为共享存储的 CMP 和共享 L2 cache 的 CMP,如图 2-19 所示。通常,早期的 CMP 采用共享 L2 cache 的 CMP 结构,即每个处理器核心拥有私有的 L1 cache,且所有处理器核心共享 L2 cache。但是共享 L2 cache 会引起不同核之间的共享竞争,因此发展到为每个内核设置独立的 L2 cache。



(a) 共享存储 CMP



(b) 共享 L2 cache CMP

图 2-19　多核处理器系统的组织结构

根据芯片上集成的多个处理器核心是否相同，CMP又可分为同构CMP和异构CMP。同构CMP大多数由通用的处理器组成，多个处理器内核相同，地位对等，执行相同或者类似的任务。当前Intel公司和AMD公司主推的多核处理器，就是同构的多核处理器。同构CMP的典型代表是美国斯坦福大学在1996年研制的Hydra CMP，Hydra集成了4个MIPS R3000处理器核，如图2-20所示。异构CMP多采用“主处理核＋协处理核”的设计，除含有通用处理器作为控制、通用计算之外，针对特定的应用集成DSP、ASIC和VLIW等协处理器来提高计算的性能。IBM、索尼和东芝联手推出的Cell BE处理器就是异构CMP的典型代表，相关内容详见本书2.4节。

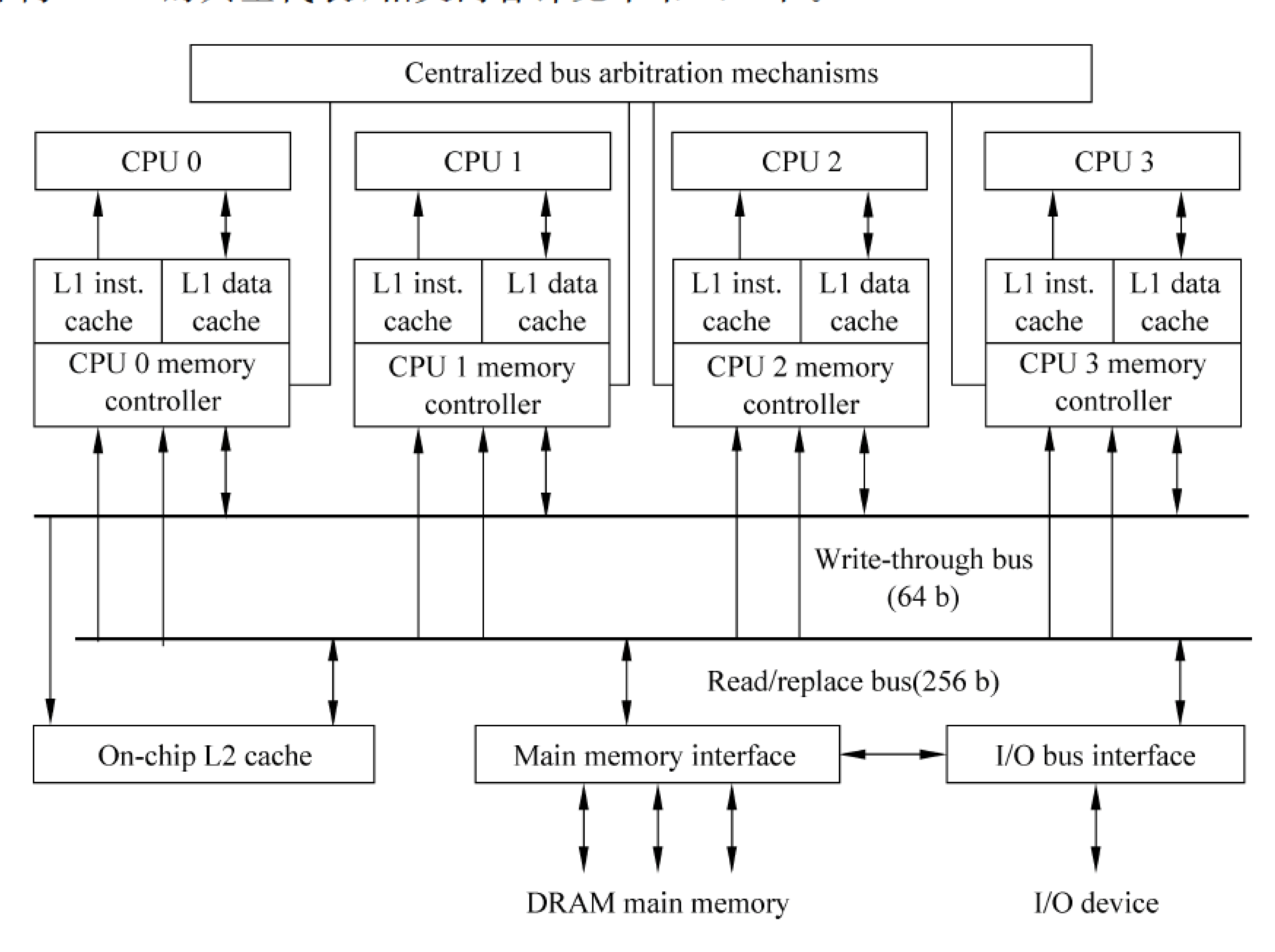


图2-20 斯坦福的Hydra CMP结构示意图

2. 多核处理器发展历史及典型架构

早在1989年，Intel公司发布了80486处理器，它是将80386、80387以及一个8KB的高速缓存封装在一个芯片中。因此，在一定意义上，80486可看作多核处理器的原始雏形。而多核处理器最直接的发展始于IBM公司的64位Power 4处理器（于2001年第二季度推出，定位于高端服务器市场），Power 4体现了当时超大规模集成电路技术和微处理机设计技术的最高水平。

Power 4是首款采用多核设计的服务器处理器，其处理器架构如图2-21所示。它采用先进的七层金属0.18μm铜互连制造工艺，在400mm² 的管芯上集成了1亿7千万个

晶体管，芯片含有两个时钟频率为 1GHz 的完整 CPU，它们共享芯片内的 3 个 L2 cache（总容量为 1.5MB），128MB 的 L3 cache 与主存控制逻辑被集成在管芯内。

这之后，Sun 公司和 HP 公司分别推出了基于双核架构的 UltraSPARC 和 PA-RISC 芯片，但此时，双核心处理器架构还都是高端的 RISC 架构，直到 Intel 和 AMD 相继推出了各自的双核心 CISC 处理器，双核处理器才真正进入主流的 x86 架构。

1）Intel 多核处理器

2005 年，Intel 公司首先发布了采用双核心设计的 Pentium D，正式进入 x86 处理器的多核心时代。Pentium D 采用两个 Prescott 内核和 1MB×2 的 L2 cache 方案。但是，Pentium D(如图 2-22 所示)还谈不上是一套完美的双核架构。因为，Intel 只是将两个完全独立的 CPU 核心集成到同一芯片上，通过同一条前端总线与芯片组相连。两个核心之间缺乏必要的协同和资源共享能力，而且还必须频繁地对 L2 cache 作同步化刷新操作，以避免两个核心的工作步调出现问题。同时，双核心的 Pentium D 的发热量、功耗均非常大，普通散热器根本无法胜任，这在当时也带动了散热器产业。尽管后来发布了 65nm 的 Pentium D 系列，发热量得到改善，频率有所提高，但其性能仍比不上 AMD 的 Athlon 64 X2 与 Athlon 64 X2 FX。

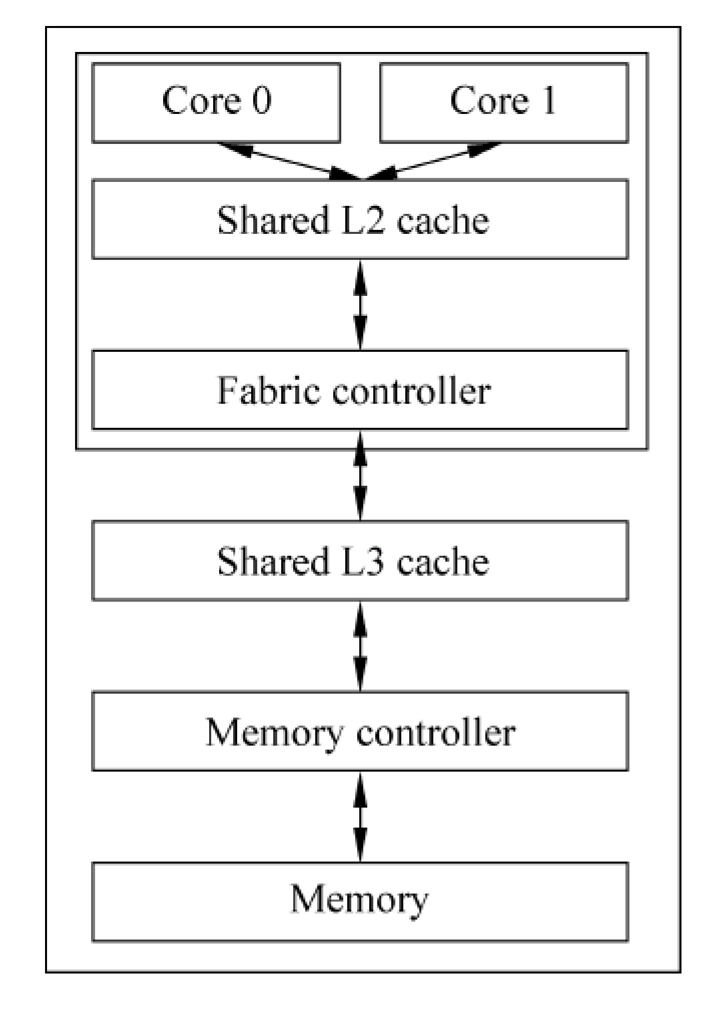



图 2-21　IBM Power4 处理器架构

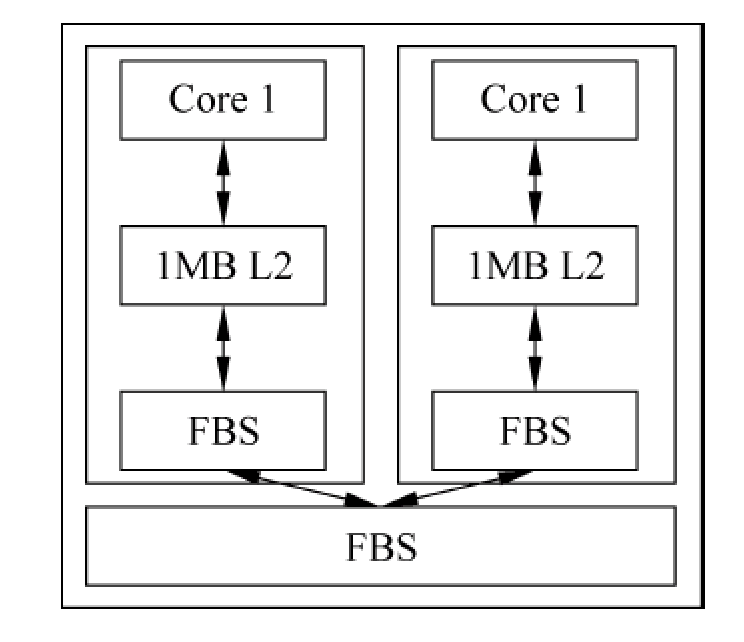


图 2-22　Intel Pentium D 处理器架构

2006 年初，Intel 公司发布了 Core 双核心处理器。Intel Core 是在 Yonah 微架构基础上改进而来的新一代微架构。其最显著的变化在于，为了提高两个核心内部数据的交换效率，Intel Core 采用了共享式 L2 cache 设计，2 个核心共享容量高达 4MB 的 L2 cache。其内核采用较短的 14 级有效流水线设计，每个核心都内建 32KB 的指令 L1 cache 与 32KB 的数据 L1 cache，2 个核心的数据 L1 cache 之间可以直接传输数据。每个核心

内建 4 组指令解码单元，支持微指令融合与宏指令融合技术。在每个时钟周期内，最多可以解码 5 条 x86 指令，并拥有改进的分支预测功能。每个核心内建 5 个执行单元子系统，执行效率高，支持 EM64T 与 SSE4 指令集。

2006 年 7 月，Intel 公司发布了 Core 2 Duo 双核心处理器，如图 2-23(a)所示。Core 2 Duo 是一个跨平台的构架体系，包括服务器、桌面、移动三大版本。其中，服务器版的开发代号为 Woodcrest，桌面版的开发代号为 Conroe，移动版的开发代号为 Merom。Core 2 Duo 在单个芯片上封装了 2.91 亿个晶体管，在功耗降低 40%的同时，能够提供满足当前和未来应用所需的性能要求。

2006 年 11 月 14 日，Intel 公司在全球率先推出了四核心处理器 Core 2 Extreme，如图 2-23(b)所示。Core 2 Extreme 包括桌面和服务器两个版本，其中前者代号为 Kentsfield，主攻高端桌面。后者代号为 Clovertown，主攻双路服务器和工作站领域。由此，PC 业进入四核心时代。虽然 Intel 抢先迈入了四核心时代，但 Kentsfield 和 Clovertown 并非是在一个晶片上集成全部四颗核心，而是将两个独立的双核心处理器封装在一起。

2007 年，Intel 公司发布了 Core 2 Quad 处理器，如图 2-23(c)所示。Core 2 Quad 拥有四个处理核心，采用了 45nm 工艺制程，其优点在于进一步压缩了处理器线路与晶体管的尺寸，在获得 40%性能提升的同时，功耗降低了 40%，这使得 Core 2 Quad 更加适合高清娱乐的应用，同时具有更低的功耗。

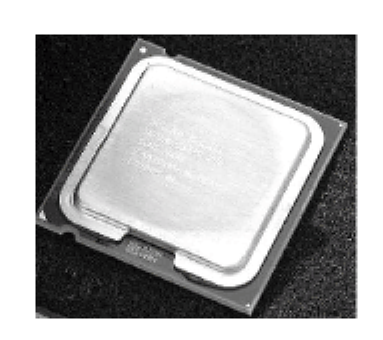




(a) Intel Core 2 Duo



(b) Intel Core 2 Extreme

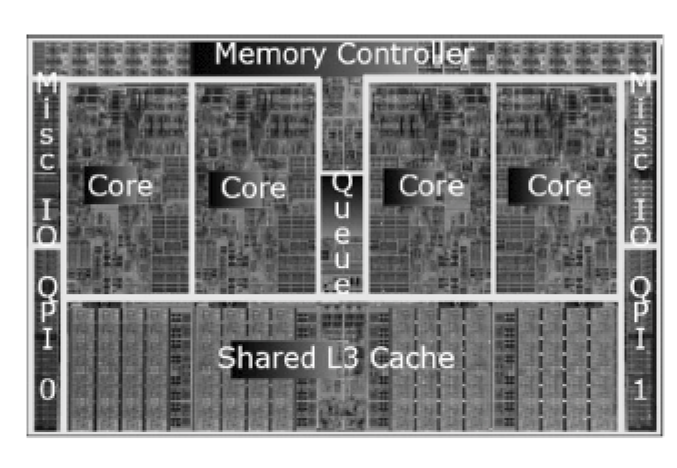


(c) Intel Core 2 Quad

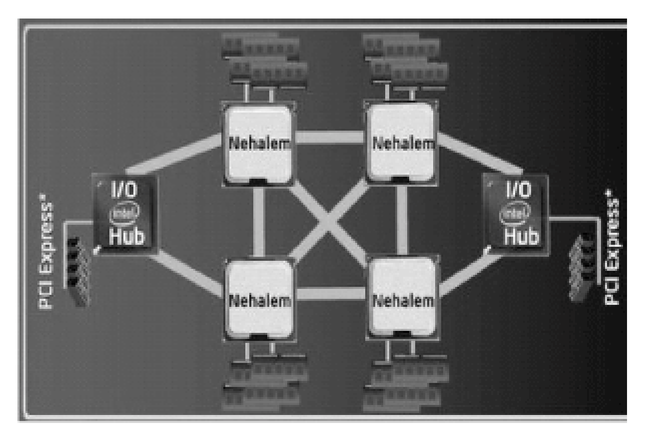
图 2-23 Intel Core 2 系列处理器

2008 年 11 月，Intel 公司发布了新一代处理器 Core i7，如图 2-25(a)所示。Core i7 虽然采用的是全新 Nehalem 架构，但 Nehalem 是建立在 Core 微架构基础上通过大幅增强与改进而来。Core i7 采用 LGA 1366 接口，增添了超线程(HT)、L3 cache、TLB 和分支预测的等级化、集成内存控制器(IMC)、QPI 总线和 DDR3 等技术。

Core i7 采用全新 L3 cache 设计，如图 2-24(a)所示。L1 cache 和 L2 cache 为内核缓存，具有超低延迟。其中，L1 cache 由 32KB 的指令缓存与 32KB 的数据缓存组成。Core i7 的 L2 cache 和 Core 2 的 L2 cache 并不相同，Core i7 的 L2 与 L1 均为内核缓存。L3 cache 采用共享式设计，被片上所有内核共享，容量为 8MB。



(a) Core i7 体系结构



(b) Core i7 QPI 总线连接方式

图 2-24　Intel Core i7 处理器体系结构与总线连接方式




(a) Intel Core i7



(b) Intel Core i5



(c) Intel Core i3

图 2-25　Intel Core i 系列处理器

Core i7 的 Nehalem 架构最大的改进在前端总线(FSB)上,传统的并行传输方式被彻底废弃,转而采用类似于 PCI Express 串行点对点传输技术的通用系统接口(CSI),Intel 称之为 QuickPath Interconnect(QPI)总线技术,如图 2-24(b)所示。QuickPath 的传输速率为 6.4GB/s,一条 32 位的 QuickPath 带宽能达到 25.6GB/s。QuickPath 的传输速率是 FSB 1333MHz 的 5 倍,虽然前者的数据位宽较窄,但其传输带宽仍然是后者的 2.5 倍。更高带宽的 DDR3 内存加上三通道技术的引入,使得 FSB 的传输带宽已不能满足要求而成为系统瓶颈,因此引入全新的 QPI 总线势在必行。通过 QPI 总线,可有效降低处理器和各个硬件之间数据传输的延迟,能有效地提高系统的性能。

Core i7 拥有集成内存控制器(Integrated Memory Controller,IMC),而且支持三通道的 DDR3 内存,运行在 DDR3-1333(支持 XMP 技术的内存更可以 1600MHz 的频率运行),内存位宽从 128 位增加到 192 位,总的峰值带宽可达到 32GB/s,为 Core 2 的 2~4 倍。采用集成内存控制器之后,处理器就能直接与物理存储器阵列相连接,从而在很大程度上减少了内存延迟的现象。

Core i7 支持睿频加速技术(Turbo Boost),这是一种基于 Nehalem 架构的电源管理技术,通过分析当前 CPU 的负载情况,智能关闭一些不使用的核心,把能耗留给正在使用的核心,并使它们运行在更高的频率,进一步提高性能;相反,当程序需要多个核心时,将开启相应的核心,重新调整频率。这样,在不影响 CPU 的 TDP(热设计功耗)情况下,能够将核心的工作频率调得更高。

Core i7 支持完整的流式单指令多数据流扩展(Streaming SIMD Extensions 4,SSE 4)指令集。SSE 4 共包含 54 条指令,其中的 47 条指令已在 45nm 的 Core 2 上实现,称为 SSE 4.1。SSE 4.1 指令的引入,进一步增强了 CPU 在视频编码/解码、图形处理以及游戏等多媒体应用上的性能。其余的 7 条指令在 Core i7 中也得以实现,被称为 SSE 4.2。SSE 4.2 是对 SSE 4.1 的补充,主要针对 XML 文本的字符串操作、存储校验 CRC32 的处理等。

Intel 公司早在 2008 年底就推出了 Core i7 处理器,虽然其性能较强,但价格也非常昂贵,因此在当时并未被普通用户普遍接受。直到 2009 年 9 月 6 日,Intel 公司正式对外发布了面向中高端用户的 Core i5 处理器,如图 2-25(b)所示。Core i5 是基于 Nehalem 架构的双核处理器,采用集成内存控制器与 L3 cache 模式,L3 cache 的容量达到了 8MB,采用成熟的 DMI(Direct Media Interface),只支持三通道的 DDR3 内存,支持 Turbo Boost 等技术,不支持超线程技术,使用 LGA1160(后改为 LGA1156)接口。

2010 年初,Intel 公司正式发布了 Core i3 处理器,如图 2-25(c)所示。Core i3 可看作 Core i5 的进一步精简版,最大的特点是整合了 GPU(图形处理器)。Core i3 的 CPU 部分采用双核心设计,通过超线程技术可支持四个线程,L3 cache 的容量为 4MB,支持内存控制器、双通道、智能加速、超线程等技术,同样采用 LGA1156 接口。

2) AMD 多核处理器

2006 年,AMD 也推出了双核 Athlon 64 X2 处理器,如图 2-26 所示。但与 Intel 不同的是,AMD 早在设计 K8 架构时就考虑到了集成双核的可能性,而且为了构建多处理器的弹性互连架构,为 K8 核心增加了一个专门与其他 CPU 通信的任务指派单元。这样,当 AMD 要开发双核产品时就显得比较顺利。

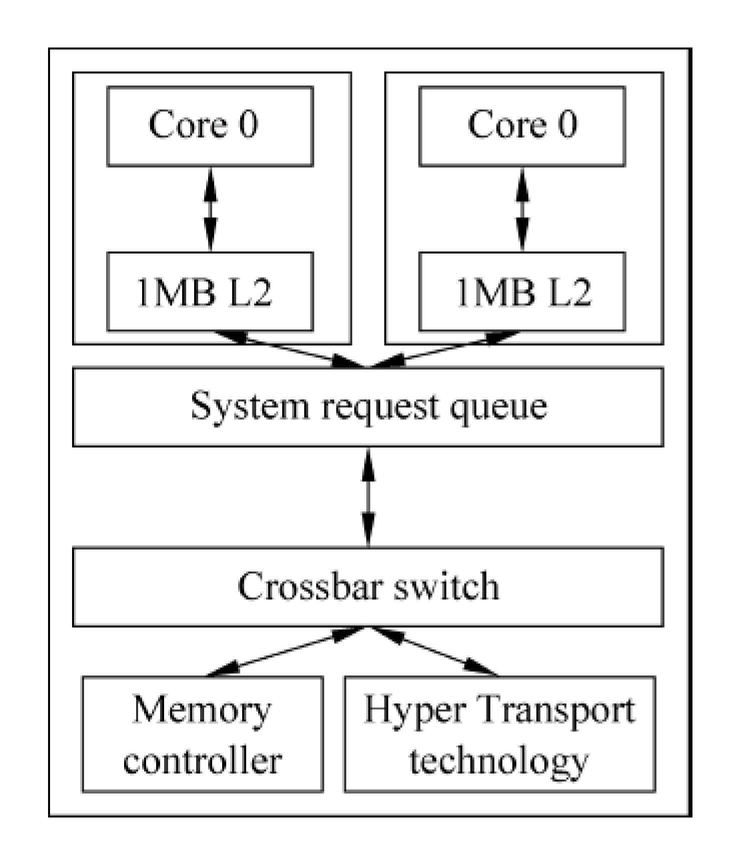


图 2-26 AMD Athlon 64 X2 架构图

Athlon 64 X2 采用 512KB/1MB×2 的独占式 L2 cache 设计,两个核心共享 Hyper Transport。Hyper Transport 技术通过消除 I/O 瓶颈来提高系统带宽,降低系统延迟增强了系统的总体性能。Athlon 64 X2 处理器内部整合了 DDR 内存控制器,全面集成的 DDR 内存控制器为处理器和主板提供直接连接,有助于提高内存的访问速度。Athlon 64 X2 还采用了系统请求队列(System Request Queue,SRQ)技术,每个内核都将其请求放在 SRQ 中,当获得资源后,请求会被发送到相应的请求内核,所以其缓存数据的一致性不需要通过北桥芯片,在处理器内部就可以直接完成。在性能、功耗、发热量等方面,Athlon 64 X2 系列几乎都要优于 Intel 的 Pentium D。但 Athlon64 X2 处理器的价格却要比 Pentium D 高出不少,这使得用户更偏向于 Pentium D 平台。

2007 年 8 月，AMD 推出了代号为 Barcelona 的真四核 AMD Opteron 服务器处理器，如图 2-27(a)所示。AMD Opteron 处理器采用了直连架构，通过减小延迟来改进性能。采用 AMD PowerNow 技术，支持双动态电源管理与智能预取技术，集成了 DDR2 DRAM 控制器。

2007 年 11 月，AMD 推出基于 AMD K10 架构的 Phenom X4 处理器，如图 2-27(b)所示。首次推出的型号为 Phenom 9500 和 9600，主频 2.2/2.3GHz，L2 cache 与 L3 cache 的容量均为 2MB，热设计功耗 95W，HT 的总线频率 3.6GHz。它们采用的是 65nm 制造工艺，Socket AM2＋接口封装，集成了 4.5 亿个晶体管，核心的面积为 $285mm^2$。

2008 年 3 月，AMD 推出了 Phenom X3 Triple-Core 处理器，该处理器基于 AMD K10 架构，更好地修复了 TLB 错误，平台稳定性的表现更加理想。Phenom X3 Triple-Core 还支持 200MHz 的外频，内建 3×512KB 的 L2 cache 和 2MB 的共享 L3 cache，全面提供 Hyper Transport 3.0 总线、SSE、SSE2、SSE3、SSE4A 多媒体指令集以及 x86-64 指令集。为了核心技术的一致性，在大部分的核心参数特性上，Phenom X3 和更高端的 Phenom X4 没有太大的区别。

2008 年底，AMD 公司推出了代号为 shanghai 的 45nm 四核 AMD Opteron 服务器处理器，如图 2-27(c)所示。该系列处理器大幅提高了时钟频率，L3 cache 的容量从 Barcelona 的 2MB 提高到 6MB，支持 DDR2-800 内存，性能比上一代 Barcelona 提高了 35%，功耗却降低了 35%。

2009 年 6 月，AMD 公司推出了代号为 Istanbul 的六核 AMD Opteron 服务器处理器，如图 2-27(d)所示。六核 AMD Opteron 服务器处理器采用既有平台基础，搭配低成本、省电的 DDR2 内存结构，协助减少系统升级成本。支持 Virtualization 技术(AMD-V)与 AMD-P 电源管理功能。与前一代的四核处理器相比，每瓦性能提高了 34%。

2009 年初，AMD 公司推出了采用 45nm 制造工艺的 Phenom Ⅱ X4 处理器，如图 2-27(e)所示。首批发布了 Phenom Ⅱ X4 940/920 与 Phenom Ⅱ X3 710/720，前者采



(a) AMD Opteron Barcelona



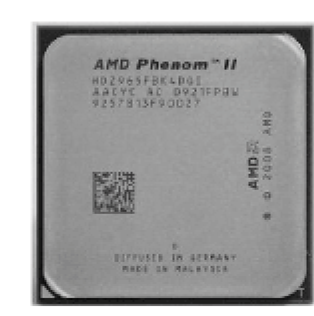
(b) AMD Phenom



(c) AMD Opteron shanghai



(d) AMD Opteron Istanbul



(e) AMD Phenom Ⅱ



(f) AMD Athlon ⅡX2

图 2-27　AMD 多核处理器系列

用 AM2＋封装，而后者采用的是原生 AM3 封装技术。在性能方面，Phenom Ⅱ X4 940/920 基本可以与 Intel 上一代准旗舰 Q9400 系列和 Q9500 系列抗衡，但售价相对更实惠。

2009 年 6 月，AMD 公司将 45nm 技术用于全新的主流处理器设计，推出了 AMD Athlon II X2 处理器，如图 2-27(f)所示。首批发布了面向普通用户的 AMD Athlon II X2 250 处理器和面向发烧友与超频玩家的 AMD Phenom Ⅱ X2 550 黑盒版处理器。AMD Athlon II X2 处理器的热设计功耗(TDP)为 65W。在运行基本任务、高负载工作和闲置状态时，分别最多可节能 50%、40%和 50%。而诸如 AMD Phenom Ⅱ X2 550 等 AMD 黑盒版处理器，能够帮助用户进行控制并最大限度地发挥性能。

3) IBM 的多核处理器

2004 年，IBM 公司推出了 Power 5 处理器，适用于 64 位计算系统，同时还向下兼容 32 位系统。每个处理器包含 2 个核心，采用 SMT(Simultaneous Multi-threading)技术，每个核心可支持 2 个线程的并发执行(即对非冲突指令，每个时钟周期都可保证两个线程各有 1 条指令同时执行)。因此，从操作系统角度来看，一个 Power 5 处理器可虚拟成 4 个 CPU。这种多线程机制可在运行时进行动态地配置。每个核心有自己独立的 L1 cache，2 个核心共享 L2 cache。可添加 L3 cache，但处理器内只集成了 L3 控制器，存储体则位于处理器的外部。内置的内存控制器有两个单向总线与内存相连，分别执行读写操作，读数据的总线为 16 位宽，最高带宽为 17.1GB/s，写数据的总线为 8 位宽，最高带宽为 8.5GB/s。每个核心在每个时钟周期内可同时执行 8 条指令，指令可乱序发射。该款处理器支持指令级并行、任务级并行、内存级并行以及核心级并行。

2007 年，IBM 公司推出了 Power 6 处理器。其核心和缓存结构与 Power 5 相同，但增加了一个内存控制器，把读操作与写操作彻底分开，使得 16 位读操作的带宽可达 51.2GB/s，8 位写操作的带宽可达 25.6GB/s，增加了支持所谓短向量操作 SIMD 指令的 AltiVec 机制，此外还增加了专门的检查点和重启动电路以提高错误检测和恢复功能。该款 CPU 可支持指令级并行、SIMD 指令、任务级并行、内存级以及核心级并行。

4) Sun 的多核处理器

2005 年 Sun 公司推出了 UltraSPARC T1 处理器，如图 2-28(a)所示。UltraSPARC T1 包含 8 个核心，每个核心可支持 4 个线程。因此，从操作系统角度来看，UltraSPARC T1 可被看作 32 个虚拟 CPU。其主要的应用目标为处理高吞吐率负载，如 Web 服务、事务处理等。所有 CPU 核心共享一个浮点计算部件，因此此款 CPU 并不适合高性能计算。

2007 年 Sun 公司推出了 UltraSPARC T2 处理器，如图 2-28(b)所示。其核心结构与 UltraSPARC T1 类似，每个核心可支持 8 个线程并发，为操作系统调度方便，8 个线程被划分为 2 个静态组。从操作系统角度来看，UltraSPARC T2 可被看作 64 个虚拟 CPU。T2 的每个核心分别配置了独立的浮点单元，但在每个时钟周期内仅允许一个核心使用。

核心内部有私有的 L1 cache，所有的核心共享 L2 cache，处理器内置 4 个内存控制器，可提供最高 42.6GB/s 的读操作带宽和 21.3GB/s 的写操作带宽。该款处理器可支持任务级并行、内存级并行以及核心级并行。




(a) Sun UltraSPARC T1



(b) Sun UltraSPARC T2

图 2-28　Sun UltraSPARC T 系列处理器

5）其他专用多核处理器

这里列出的并不是市场化、产品化特别成功的处理器，但这些技术可能代表了未来处理器系统设计的一些重要发展趋势。

（1）Cray XMT/Treadstorm

Threadstorm 多核处理器可支持多达 128 个线程，每个 CPU 核心内设 32 个通用寄存器。在 1 个时钟周期内，处理器即可切换线程。该款处理器擅长运行那些数据与指令的局部性较差的并行程序，那些在分布式存储系统上性能较差的并行程序可在该系统上取得较高的性能。每个 Cray XMT 系统可包含多达 8192 个 Threadstorm 处理器，每个处理器峰值计算速度为 1.5GFLOPS。其处理器接口与 AMD Opteron 兼容。

（2）IBM BlueGene/P

IBM BlueGene/P 多核处理器是 IBM 公司专门为 BlueGene/P 系统开发的专用处理器，属于 System-on-chip 结构。它集成了 4 个 32 位 PowerPC 450 核心、1 个内存控制器、1 个网卡以及 1 个超立方体互联网接口，4 个核心在片内有共享缓存，可运行传统的共享存储的程序（如 Pthreads 或 OpenMP 等）。各核心的频率为 850MHz，在一个时钟周期内可执行 4 条指令。因此，每个 CPU 核心的峰值计算速度为 3.4GFLOPS。它的峰值速度虽然不及 Intel 的 Core 2（每个核心的频率为 4GHz，峰值计算速度为 12GFLOPS），但其功耗却大大低于 Intel Core 2 CPU（每个处理器 4 个核心的功耗仅 16W），因此非常适合安装在 BlueGene/P 这样采用大规模 CPU 的系统中。

（3）IBM Cyclops-64

Cyclops-64 处理器内置 80 个核心，每个 CPU 核心都支持 2 个线程的并发执行，2 个线程共享 1 个浮点运算单元（在一个时钟周期内，每个浮点运算单元可执行 2 条浮点运算）。以 500MHz 主频运行，每个处理器可提供 80GFLOPS 的峰值计算能力。处理器内置了内存控制器、网卡以及超立方体的互联网接口（包括 CPU 核心在内的所有这些逻辑电路都通过一个内置于处理器内部的 96 口 7 级非阻塞交叉开关互连在一起）。每个

CPU 核心还有一个私有内存空间，可用软件配置的办法，来指定该私有内存空间究竟是核心私用还是作为整个处理器的全局共享内存，或者两者的组合。该款处理器还有一个特别的地方，即把存储层次的物理结构直接暴露给上层应用程序进行控制，因此不提供传统通用处理器所拥有的虚拟内存机制，而仅提供了一种简单的基于段的内存保护机制作为内核级程序的保护措施。

（4）SiCortex

SiCortex 集成了 6 个 64 位的 MIPS 核心、1 个内存控制器、1 个高性能网卡、1 个千兆以太网卡和 1 个 PCI-E 接口（也就是说，实际上集成了一个机群结点所需的除内存、硬盘之外的所有部件），每个核心的峰值计算速度约为 1GFLOPS。在一个配置 1 个处理器和 4GB 内存的结点中，6 个核心共享两个内存通道共计 10.6GB/s 的带宽，功耗为 12W 左右。目前最大规模的 SiCortex 系统为在 1 个机柜中安装了 972 个结点（5832 核心）。

（5）ClearSpeed CSX600

ClearSpeed CSX600 处理器包含了 96 颗核心，每个核心拥有 128KB 本地内存，所有核心共享一个内置的 128KB 片内内存，峰值计算速度可达 33GFLOPS，功耗约 10W。CSX600 执行 SIMD 数据并行指令，移植了若干数学库/工具供并行程序调用。该款处理器可作为通用 CPU 的加速部件，以外接卡的形式集成到系统中。

（6）Tilera Tile64

Tilera Tile64 是以商业化推广 MIT 的 RAW 为目标设立的新兴公司的产品。RAW 处理器内集成 64 个核心，以条块化（tiled）方式使用 mesh 网络进行互连，集成了 1 个 10GB 的以太网卡、1 个 PCI-E 接口、4 个内存控制器和 1 个软件可配置的 I/O 接口。64 个核心可被划分为若干个 cache 关联组，每个组均可独立运行操作系统。该款处理器尚未集成浮点计算部件，因此尚不支持大规模的科学计算，但在一些嵌入式系统中有很好的应用，如数字视频的处理、网络路由等。64 个核心的功耗约为 15～22W，提供每秒约为 1.9 万亿次操作的处理能力。

（7）SPI Storm-1

SPI Storm-1 是为商业化推广 Stanford 的 STREAM 处理器而设立的新兴公司所开发的产品。该款处理器也面向嵌入式应用市场，目前在高性能数值计算领域还没有太强的竞争力。一个处理器内包含一个通用的 MIPS 核心和一个多道（Multi-Lane）STREAM 单元。STREAM 单元受运行嵌入式 Linux 的 MIPS 核心管理，它不运行任何操作系统。STREAM 编程的关键在于将求解的问题分解为若干数据流，并为之定义相关的执行模块以及模块之间的关联关系。其原理与向量机的指令类似，只是扩展成可为一组数据流定义一个操作集合。STREAM 编译器负责产生任务之间的关联图与执行策略。目前最新款处理器为 STREAM-1 SP16HP-G220，其主频为 700MHz，集成了 16 个 Lane，峰值计算能力为 2.24×10^{11} 次乘法操作/秒。虽然目前还不支持浮点计算，但 SPI

Storm 处理器预留了集成浮点处理部件的机制，为在将来进入高性能数值计算领域做好了准备。

(8) Ambric Am2045

Ambric Am2045 也是一个以开发大规模并行处理器阵列(Massively parallel processor arrays，MPPA)为目的的新兴商业公司推出的产品。该处理器从开发伊始就把支持消息传递编程模型作为其设计的核心理念。可以说 MPPA 是将传统的 MPP 大型机进行芯片化的一种尝试，只是使用点对点的互联网将处理器核心(在原来 MPP 系统中，通过共享内存进行通信)加以连接，新的处理器中各个核心可混合使用分布式内存和共享内存。其原型产品 Am2045 处理器集成了 336 个 32 位 RISC 核心，每个核心具有私有的 2KB 内存。各个核心可独立运行程序，通过片内内置的软件可配置多级互联网与其他核心通信，可使用 Java 或汇编进行编程。处理器主频为 350MHz，峰值计算速度为 6×10^{10} 次乘法运算/秒，目前尚不具备浮点运算部件，因此在高性能计算领域的应用还不成熟。

2.2.4　多核处理器关键技术

虽然单芯片多处理器架构利用多处理器优势所带来的诸多好处，让处理器的性能成倍地增加。但随之而来的是将原来系统级的一些问题引入了处理器内部。CMP 的关键技术如下所示。

1. 核心结构的选择

目前多核处理器的核心结构主要有同构和异构两种。同构结构采用对称设计，原理简单，硬件上较易实现。当前主流的双核、四核处理器基本上都采用同构结构。同构设计的问题在于：

- 随着核心数量的不断增多，如何保持各个核心的数据一致性；
- 如何满足核心的存储访问和 I/O 访问需求；
- 如何选择一个各方面性能均衡、面积较小以及功耗较低的处理器；
- 如何均衡若干处理器的负载和任务协调等。

与同构结构相比，异构的优势是通过组织不同特点的核心来优化处理器内部结构，实现处理器性能的最优化，而且能有效地降低功耗。但是异构结构也存在如下难点：

- 搭配哪几种不同的核心，核心之间任务如何分配以及如何实现；
- 结构是否具有良好的扩展性，还是受到核心数量的限制；
- 如何设计和实现处理器指令系统，因为不同核心所用的指令系统对多核处理器的实现也是很重要的。不同核心是采用相同的指令系统还是不同的指令系统，能否运行操作系统等，也是需要考虑的内容。

2. 多级缓存设计与一致性问题

随着半导体工艺的发展，在 CMP 系统中处理器和主存间的速度差距越来越突出，因此必须使用多级缓存来缓解。目前有共享 L1 cache 的 CMP、共享 L2 cache 的 CMP 以及共享主存的 CMP。但是在 CMP 结构中，共享缓存或私有缓存孰优孰劣，需不需要在一块芯片上建立多级缓存以及建立几级缓存，每一级缓存的大小是多少，缓存对整个芯片尺寸、功耗、布局的影响等问题都对 CMP 的性能以及运行效率等有很大的影响，须认真研究和探讨。

另一方面，多级缓存又引发了一致性问题。采用何种缓存一致性模型和机制都将对 CMP 整体性能产生重要影响。在传统多处理器系统结构中广泛采用的缓存一致性模型有顺序一致性模型和弱一致性模型等。与之相关的缓存一致性协议主要有基于总线监听、基于目录和面向编译的缓存一致性协议等。目前的 CMP 系统大多采用基于总线监听的缓存一致性协议。

3. 核间通信技术

CMP 处理器的各 CPU 核心执行各自的程序代码，这些程序之间有时需要进行数据共享与同步，因此其硬件结构必须支持核间通信。高效的通信机制是 CMP 处理器高性能的重要保障。

目前比较主流的片上高效通信机制有如下两种：

- 基于总线共享的缓存结构，以斯坦福大学的 Hydra 处理器为代表；
- 基于片上互连的结构，以麻省理工学院的 RAW 处理器为代表。

基于总线共享的缓存结构是指每个 CPU 内核拥有共享的 L2 或 L3 cache，用于保存比较常用的数据，并通过连接核心的总线进行通信。这种系统的优点是结构简单，通信速度高。缺点是基于总线的结构可扩展性较差。

基于片上互连的结构是指每个 CPU 核心具有独立的处理单元和缓存，各个 CPU 核心通过交叉开关或片上网络等方式连接在一起。各个 CPU 核心间通过消息通信。这种结构的优点是可扩展性好，数据带宽有保证。缺点是硬件结构复杂，软件改动较大。

未来的核间通信技术究竟采用哪种方式，还是两者结合取长补短，都是将来需要考虑的问题。例如，在全局范围采用片上网络而局部采用总线方式，来达到性能与复杂性的平衡。

4. 总线设计

在传统处理器中，缓存不命中或访存事件都会对 CPU 的执行效率产生负面影响，而总线接口单元(BIU)的工作效率会决定此影响的程度。当多个 CPU 核心同时要求访问内存或多个 CPU 核心内的私有缓存同时出现缓存不命中事件时，BIU 对这些访问请求

的仲裁机制以及对外存访问的转换机制的效率决定了 CMP 系统的整体性能。因此，寻找高效的 BIU 结构，将多核心对主存的单字访问转化为更为高效的 Burst(猝发)访问。同时，寻找使 CMP 处理器整体效率达到最优的一次 Burst 访问字的数量模型，以及寻找高效多端口 BIU 访问的仲裁机制，将是 CMP 处理器研究的重要内容。

5. 任务调度、中断处理和同步互斥设计

对于多核 CPU，优化操作系统的任务调度算法是保证效率的关键。任务调度算法一般有局部队列调度和全局队列调度。

- 局部队列调度是指操作系统为每个 CPU 内核维护一个局部的任务等待队列，当系统中有一个 CPU 内核空闲时，便从该核心的任务等待队列中选取恰当的任务执行，该方法的优点是任务基本上无须在多个 CPU 核心间切换，有利于提高 CPU 核心的局部缓存命中率；
- 全局队列调度是指操作系统维护一个全局的任务等待队列，当系统中有一个 CPU 核心空闲时，操作系统就从全局任务等待队列中选取就绪任务开始在此核心上执行。该方法的优点是 CPU 核心的利用率较高。目前多数多核 CPU 操作系统采用的是基于全局队列的任务调度算法。

多核的中断处理和单核有很大不同。多核的各处理器之间需要通过中断方式进行通信，所以多个处理器之间的本地中断控制器和负责仲裁各核心之间中断分配的全局中断控制器也需要封装在芯片内部。

另外，多核 CPU 是一个多任务系统。由于不同任务会竞争共享资源，因此需要系统提供同步与互斥机制。而传统的用于单核的方法并不能满足多核，需要利用硬件提供的“读-修改-写”的原子操作或其他同步互斥机制来保证。

6. 低功耗设计

半导体工艺的迅速发展使处理器的集成度越来越高，同时处理器表面温度也变得越来越高并呈指数级增长。目前，低功耗和热优化设计已经成为处理器研究中的核心问题。CMP 的多核心结构决定了其相关的功耗研究是一个至关重要的课题。

低功耗设计是一个多层次问题，需要同时在操作系统级、算法级、结构级、电路级等多个层次上进行研究。每个层次的低功耗设计方法实现的效果不同，抽象层次越高，功耗和温度降低的效果越明显。

7. 存储器墙

为了使处理器内的核心充分地工作，最起码的要求是处理器能提供与处理器性能相匹配的存储器带宽，虽然内部缓存的容量能解决一些问题，但随着性能的进一步提高，必

须有其他一些手段来提高存储器接口的带宽，如增加单个管脚带宽的 DDR、DDR2、QDR、XDR 等。同样，系统也必须有能提供高带宽的存储器。所以，处理器芯片对封装的要求也越来越高，虽然封装的管脚数每年以 20%的数目提升，但还不能完全解决问题，而且还带来了成本提高的问题，为此，怎样提供一个高带宽、低延迟的接口带宽，是必须解决的一个重要问题。

8. 可靠性及安全性设计

随着技术革新的发展，处理器的应用渗透到现代社会的各个层面，但是在安全性方面却存在着很大的隐患。一方面，处理器结构自身的可靠性低下，由于超微细化与时钟设计的高速化、低电源电压化，设计上的安全系数越来越难以保证，故障的发生率逐渐升高。另一方面，来自第三方的恶意攻击越来越多，手段越来越先进，已成为具有普遍性的社会问题。当前，提高可靠性与安全性在计算机体系结构研究领域备受注目。

9. 平衡设计原则

平衡设计原则是指在芯片的复杂度、内部结构、性能、功耗、扩展性和部件成本等各个方面做一定的权衡，在设计过程中要从整体结构的角度去权衡各个具体的结构问题，不能为了单纯地获得某一方面的性能而导致其他方面的问题。在多核处理器设计工程中，项目人员需要坚持平衡设计的原则。因为，往往在减少一个方面问题的同时又增加了另一个方面的问题，所以在设计过程中要仔细权衡对某些问题的解决方法，尽量采用简单、易于实现、成本低廉而且对整体性能影响不大的设计。微处理结构设计的重点不在于其中某一个细节采用多么复杂或性能表现较好的设计，而是在于整体的设计目标。当然在具体的设计中，不能只是简单的选择，应该是建立在科学的实验和模拟分析基础上来选择或平衡。因此，在多核处理器设计中，要以科学分析的数据结果为基础，坚持合理、平衡的设计原则。

10. 应用软件开发

多核处理器在利用多个核心的并行执行能力来提高处理器运算性能的同时，也给软件开发者带来了麻烦。当前的困境是众多应用并没有利用多核的性能潜力，多核的性能优势没有体现。

多核系统下的并行编程，主要是开发多核的线程级并行性，但是已有的并行编程模式、编程语言并不完全适合多核环境，不能将多核的多线程并行潜力完全发挥出来，例如 OpenMP、MPI 和并行 C 等。因此，许多研究机构和公司一方面对现有的并行编程模型和编程语言进行修改或改进。例如，改进支持共享存储结构的 OpenMP，采用 OpenMP+MPI 的混合编程模型和 Pthread 多线程编程模型等；另外，各类研究机构也正在积极研

制开发新一代的并行编程模型和并行编程语言，例如，事务存储编程模型和 Intel 公司的 Ct 编程模型等。

同时，为了将已有的串行程序部署到多核系统上，要么重新编写并行代码，要么研发面向多核结构的自动并行化工具，使得这些应用能在多核处理器系统中高效执行。

2.2.5 多核处理器未来发展趋势

多核处理器产生的直接原因是替代单核处理器，解决单核处理器发展的瓶颈，但发展多核处理器的深层次原因还是为了满足人类社会对计算性能的无止境需求。目前，阻碍多核性能向更高水平发展的问题很多，可真正束缚多核处理器发展的是低功耗和应用开发两个问题。因此，有必要在原有技术的基础上探索新的思路和方法来解决上述问题。

为了实现高性能、低功耗和高应用性，多核处理器的几种可能发展趋势如下：

(1) 在多核上将集成更多结构简单、低功耗的核心。为了满足性能的需求，通过集成更多核心来提高性能是必然选择，但是核心的结构也必须考虑。因为如果核心结构过于复杂，随着核心数量的增多，不仅不能提升性能，还会带来互连线延迟增加和功耗变大等问题。

(2) 异构多核是一个重要的发展方向。研究表明，将结构、功能、功耗、运算性能各不相同的多个核心集成在芯片上，并通过任务划分将不同的任务分配给不同的核心，让每个核心处理自己擅长的任务。这种异构组织方式比同构的多核处理器执行任务更有效率，更能实现资源的最优配置，而且能够降低整体功耗。

(3) 多核处理器将采用更高效的片上互连机制。随着处理器内核心数目的增加，传统的共享总线机制将无法保持有效性能，这就要求多核心之间能够实现高效的点对点片内互连，同时智能地将应用程序映射到这些互连拓扑上并使之高效运行，这是操作系统、编译器等系统软件面临的迫切任务。不仅 CPU 核心之间需要特殊的互连拓扑，系统内存与处理器的连接结构同样也需要相应的拓展，传统的均匀随机访问方式将被非均匀访问方式所取代。

(4) 大规模高性能可编程器件的出现，推动了现场可编程门阵列(Fieid Programmable Gate Arrays，FPGA)技术的发展。在芯片上应用 FPGA 技术有高灵活性、高可靠性、高性能、低能耗和低成本等多种优势。处理器设计人员注意到了这种优势，并将 FPGA 的可重构技术应用到多核结构上，让多核结构具备可重构性和可编程性。这种创新思路大大提高了多核的通用性和运算性能，使处理器既有了通用处理器的通用性，又有专用集成电路的高性能，使之兼具了灵活性、高性能、高可靠、低能耗等优势。

(5) 多核心平台下并行程序的开发也是一个重要的研究方向。多核心平台的出现必然使得传统的串行程序向并行和并发程序转变。在并行/并发程序开发中，最大的难点在于其行为的不确定性，特别是并行/并发访问共享资源将会在大规模多核系统中成为开发

瓶颈和性能瓶颈，而由此带来的调试、优化等方面的问题则使得应用程序开发面临更大的困难。在过去的几十年间，工业界和学术界从未间断过对提高并行程序开发效率的研究，但迄今为止还未找到一种真正有效的解决办法。可以预见，未来硬件平台的发展，必然要求系统中集成简化编程和减少错误发生的硬件逻辑，并有针对地发展相应的编程模型，这一领域已有一些初见端倪的趋势，如事务内存(Transaction Memory)以及一些硬件同步逻辑等。

本节小结

多核处理器通过采用简化单核结构、增加核心数目和片上部件等结构设计方法提高了处理器的性能，适应了工艺发展和应用的需求，逐步成为应用的主流。多核技术的进一步发展需要解决低功耗和应用开发等重要问题，而这些问题的解决是一个综合考虑的结果。总的来看，多核正向着众核方向发展，并呈现多核心、低功耗、异构和可重构等几个方面的发展趋势。虽然现阶段多核发展仍面临众多挑战，但多核技术的未来值得期待。

2.3 GPU

GPU 的全称是 Graphic Processing Unit，即图形处理单元。它的主要功能就是进行浮点运算、定点处理和着色处理。得益于游戏业的高速发展，GPU 技术的发展达到了前所未有的速度，其更新换代的时间大大小于 CPU。GPU 的功能更新非常迅速，平均半年就有新一代的 GPU 诞生，运算速度也越来越快。例如，NVIDIA 的 G80 核心的 GPU 拥有 128 个标量浮点运算单元，在计算速度方面 GPU 已走在了 CPU 的前面，并且 GPU 的价格也相对低廉，使得运用 GPU 进行科学运算具有很高的性价比。

GPU 是显卡的核心，相当于 CPU 在计算机中的作用。GPU 决定了该显卡的档次和大部分性能，同时也是 2D 显卡和 3D 显卡的区别依据。2D 显卡在处理 3D 图像和特效时主要依赖 CPU 的处理能力，称为“软加速”。3D 显卡则将三维图像和特效处理功能集成在显示芯片内，称为“硬件加速”。目前，市场上大多采用 NVIDIA 和 AMD 两家公司的。

NVIDIA 公司在 1999 年发布 GeForce 256 时，率先提出了 GPU 的概念。GPU 使显卡减少了对 CPU 的依赖，并进行原本属于 CPU 的部分工作，尤其是在 3D 图形处理时。GPU 所采用的核心技术有多边形转换与光源处理(Transform and Lighting，硬件 T&L)、立方环境材质贴图和顶点混合、纹理压缩和凹凸映射贴图、双重纹理四像素 256 位渲染引擎等，而硬件 T&L 技术可以说是 GPU 的标志。

目前，GPU 已不再局限于 3D 图形处理了，GPU 通用计算技术的发展已引起业界的不少关注。事实证明，在浮点运算、并行计算等方面，GPU 可以提供数十倍乃至于上百倍

于 CPU 的性能。GPU 通用计算方面的标准目前有 OpenCL、CUDA、ATI STREAM。其中，OpenCL(全称 Open Computing Language)是第一个开放式、免费的面向异构系统通用目的的并行编程标准，也是一个统一的编程环境。

2.3.1　GPU 概述

目前，通用计算领域通常采用的处理器是 CPU。传统意义上来说，GPU 主要负责图形渲染等图形方面的计算。

在过去 20 年间，增加处理器芯片上晶体管的数量，提高运行频率是 CPU 性能提高的主要方式。然而，从 2003 年以来，这种趋势发生了变化。不断提高的 CPU 频率带来了高功耗和高发热量问题，使得主流 CPU 的频率止步于 4GHz，并向单芯片多处理器(Chip MultiProcessors，CMP)即多核处理器方向发展。在 2005 年，Intel 和 AMD 正式向主流消费级市场推出了双核心的 CPU 产品，在 2007 年又推出了 4 核心的 CPU，按照各厂商的发展路线图，大约每 2 年单芯片上的核心数目将翻倍。伴随着并行架构的不断发展，并行算法也在不断成熟与完善。但由于市场变化和研制成本等多方面的原因，多核 CPU 的每个核心仍基于以往单核 CPU 的设计，保留了例如乱序执行等很多单核时代的复杂执行方式，使得其对科学计算等问题的计算能力提高较为有限。

GPU 最初用于固定的功能。随着时间的推移，这些图形芯片的可编程性日益增加，并开始在非图形的高性能计算领域被大量使用。在此基础上，NVIDIA 公司推出了第一款 GPU。在 1999—2000 年间，计算机科学家与诸如医疗成像和电磁等领域的研究人员，开始使用 GPU 来运行通用的计算，并发现 GPU 具备的卓越浮点性能可为众多科学应用程序带来显著的性能提升。从图 2-29 可以看出 GPU 的发展速度已经远远超过 CPU。

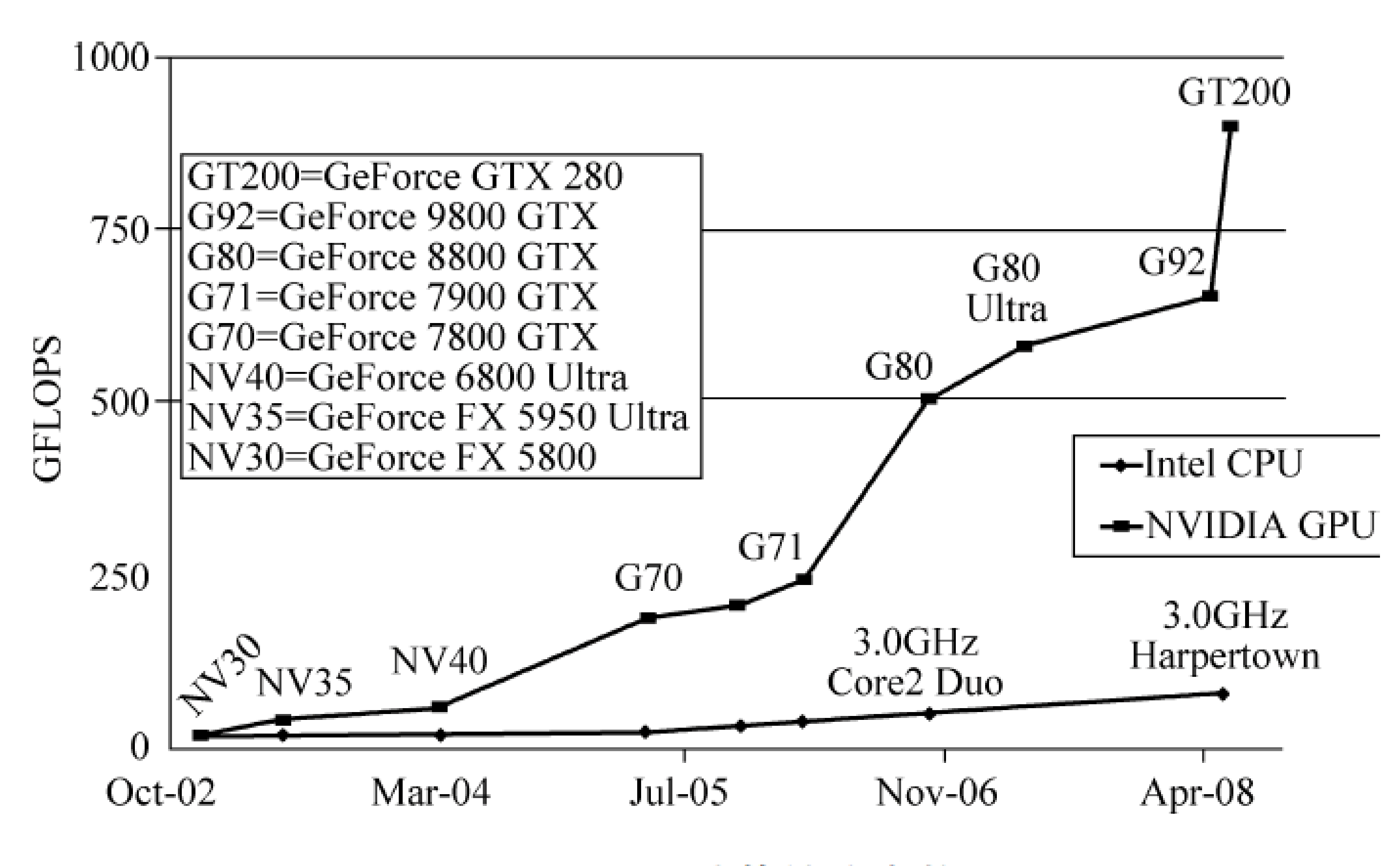


图 2-29　GPU 计算性能走势图

由图 2-29 可见，GPU 的浮点运算速度达到 CPU 的若干倍。近年来，GPU 的性能每一年就可以翻倍，大大超过了 CPU 遵循的摩尔定律（每 18～24 月性能翻倍）的发展速度。带来这种数据处理能力差别的主要原因在于，GPU 最早为并行处理大量三维计算机图形学中的顶点和像素数据而设计，但近年来为通用计算进行了一系列的改进。其并行体系结构决定了 GPU 非常擅长以并行方式运行高运算强度的应用。图 2-30 为同等市场价格的 CPU 和 GPU 的浮点运算单元数量对比图。

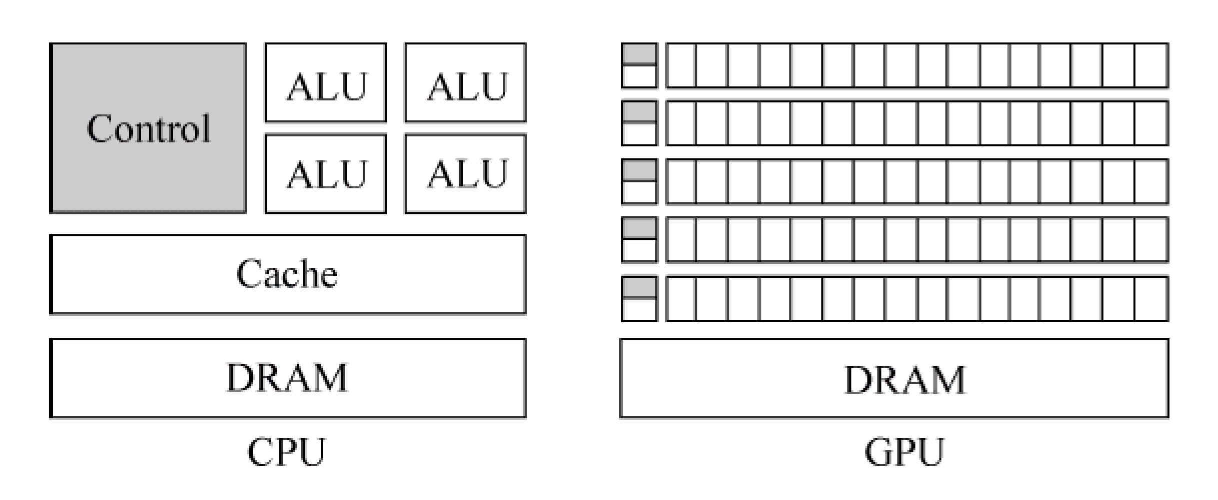


图 2-30　CPU 与 GPU 运算单元数量对比

NVIDIA 公司的 GeForce 8800 GTX 包含了 128 个流处理器，HD 2900 包含了 320 个流处理器。这些流处理器可以支持浮点运算、分支处理、流水线、单指令流多数据流（Single Instruction Multiple Data，SIMD）等技术。以 NVIDIA 公司的 G80 为例，与 G80 的 GPU 包含的 128 个核心相比，目前多核心 CPU 的核心数目明显要少得多。虽然，CPU 每个核的运算能力高于 GPU 上的每个核心，但后者凭借更多核心的并行使得其总的计算能力更强。与使用 CPU 相比，使用 GPU 进行运算具有如下优势：

(1) GPU 通常具有更大的内存带宽。例如 NVIDIA 公司的 GeForce 8800 GTX 具有超过 50GB/s 的内存带宽，而目前高端 CPU 的内存带宽则在 10GB/s 左右。

(2) GPU 具有更多的执行单元。例如 GeForce 8800 GTX 具有 128 个流处理器（StreamProcessors），频率为 1.35GHz。CPU 的频率通常较高，但是执行单元的数目则要少得多。

(3) 和高端的 CPU 相比，显卡的价格较为低廉。例如目前一块 GeForce 8800 GTX（含 512MB 内存）的价格与一个 2.4GHz 四核心 CPU 的价格相当。

当然，使用 GPU 进行运算也存在如下缺点：

(1) GPU 的运算单元数量很多，因此对不能高度并行化的工作，所能带来的帮助并不明显。

(2) GPU 目前通常只支持 32 位浮点数，且多半不能完全支持 IEEE 754 标准，有些运算的精确度可能较低。目前，许多 GPU 并没有独立的整数运算单元，因此整数运算的效率较差（NVIDIA 公司基于新一代 CUDA 架构的 Fermi 已经解决了这个问题，在后面将会提及）。

(3) GPU 通常不具有分支预测等复杂的流程控制单元，因此对于具有复杂分支的程序，效率会比较差。

总体来说，GPU 类似于流多处理器，适合一次进行大量相同的工作。CPU 则擅长处理分支较多、关联性较强的任务。

2.3.2 GPU 发展简介

从功能和架构方面来讲，GPU 经历了三个阶段的发展。

- 第一代 GPU(1999 年之前)：从 CPU 分离出部分功能，实现硬件加速。GE (Geometry Engine)作为其代表，只能起到 3D 图像处理的加速作用，不具有软件编程的特性。
- 第二代 GPU(1999—2002 年)：硬件加速得到加强并提供有限的编程性。
  在 1999 年，NVIDIA GeForce 256 将 T&L(Transform and Lighting)等功能从 CPU 中分离出来，实现了快速变换。
  在 2001 年，NVIDIA 和 ATI 分别推出了 GeForce 3 和 Radeon 8500，图形硬件的流水线被定义为流处理器，出现了顶点级可编程性，同时像素级也具有有限的编程性，但 GPU 的编程性比较有限。
- 第三代 GPU(2002 年之后)：方便的编程环境(如 CUDA)。
  在 2002 年，ATI 发布了 Radeon 9700。
  在 2003 年，NVIDIA 推出了 GeForce FX。
  在 2006 年，NVIDIA 与 ATI 分别推出了 CUDA(Computer Unified Device Architecture，统一计算架构)编程环境和 CTM(Close To the Metal)编程环境。
  2007 年，CUDA 正式发布。
  2008 年，NVIDIA 发布了支持 CUDA 1.1 的 GeForce 9 系列 GPU，以及支持 CUDA 1.3 的 GT200 GPU。NVIDIA 在 GT200 中引入了大量的重要改进，使得 GT200 不仅具有很高的处理能力和存储器带宽，用于通用计算时的可编程性和灵活性也更加出色。同年，NVIDIA 发布了 Tegra 系列产品，正式进入移动处理器市场。

随着 GPU 可编程性的不断增强，特别是 CUDA 等编程环境的出现，使得 GPU 通用计算编程的复杂性大幅度降低。由于可编程性、功能、性能的不断提高和完善，GPU 已演化为一个新型的可编程的高性能计算资源。目前，NVIDIA 正在大力推广用于通用计算领域的 GPU，GPU 已经开始全面向通用计算的方向发展。

2.3.3 GPU 硬件架构

GPU 通常用于图形处理领域，在此将主要介绍用于通用计算领域的 GPGPU。通过

单指令多数据指令类型来支持数据并行计算。在单指令多数据流的结构中，单一控制部件向每条流水线分派指令，同样的指令被所有处理部件同时执行。例如，GPU 包含了多组 SM(Streaming Multiprocessor)，每组处理器有多个 SP(Streaming Processor)，但每组处理器只包含一个指令单元(Instruction Unit)。

以 R600 为例，DPPA 是真正执行通用计算的功能单元，如图 2-31 所示。

- R600 的数据并行阵列拥有 4 个 SIMD 引擎，它们同时处理一个内核程序；
- 每个 SIMD 引擎又由 16 个线程处理器组成，这 16 个线程处理器共用 1 个 PC，所以它们之间是完全同步执行的；
- 线程分配处理器是一个超长指令字处理单元，它包括 4 个标量计算核和 1 个超级计算核；
- 1 个线程处理器通过阻塞多线程的方式同时运行 4 个线程，如图 2-31 所示。

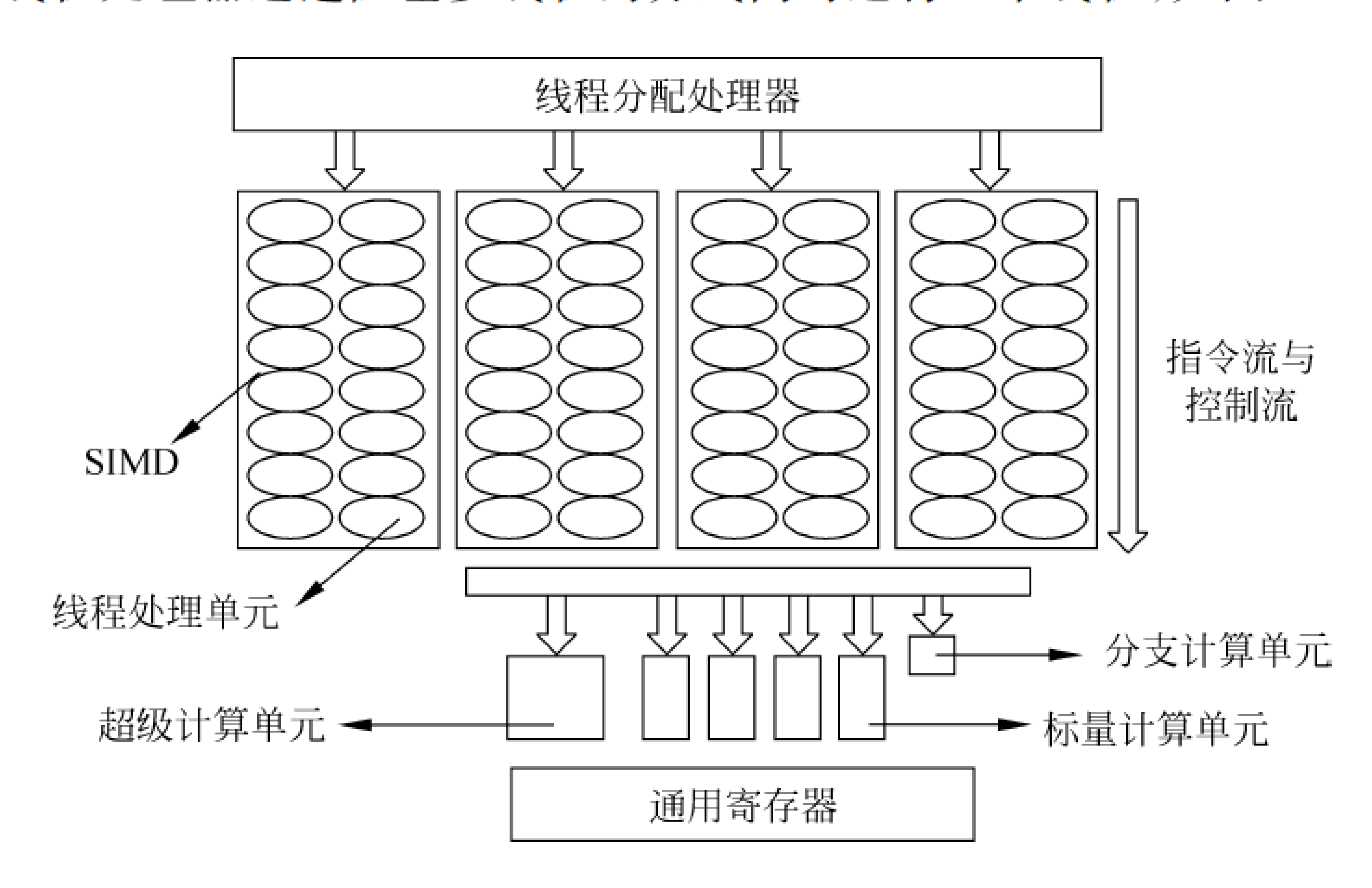


图 2-31　数据并行处理器阵列结构

经过上述分析可以发现，R600 拥有 320(即 4×16×5)个计算核，可以同时运行 256(即 4×16×4)个线程，如此的计算核规模和线程规模显然可以提供较为强大的计算能力。

通过一个简单的例子，将两个包含有 1000 个元素的数组进行相加，来说明 CPU 执行与 GPU 执行的差别。如果使用 CPU 进行处理，将进行不断的循环操作，把两个数组中的对应元素相加以产生新的数组元素。就这个例子来说，在 CPU 上执行必须进行 1000 次循环操作。如果使用 GPU 进行处理，GPU 会首先定义一个数组的加操作，并且为数组中的每一个元素生成一个加法程序的实例。然后创建 1000 个加法线程，多个线程同时执行。使用具有 240 个内核的 GTX 280 仅需要 5 个时钟周期就可完成。

GPU 通用计算方面的编程环境目前有 OpenCL、CUDA、ATI STREAM。其中，OpenCL 是第一个开放式、免费的面向异构系统通用目的并行编程环境，便于软件开发人

员为高性能计算服务器、桌面计算系统、手持设备编写高效、简洁的代码，而且广泛适用于多核心处理器(CPU)、图形处理器(GPU)、Cell 类型架构以及数字信号处理器(DSP)等其他并行处理器。在游戏、娱乐、科研、医疗等多个领域都有广阔的发展前景。AMD-ATI 以及 NVIDIA 现在的产品都支持 OpenCL。

NVIDIA 也会继续加强对包括 C 语言在内的其他语言的支持，NVIDIA CUDA C 目前还是唯一针对 GPU 的 runtime C 语言环境(是指 GPU 可直接执行该语言)。NVIDIA CUDA C 语言还会进一步发展，不断会有新的版本推出。NVIDIA CUDA C 语言将与 OpenCL、DX11 等共存。本书的第 3 章将会以 CUDA 为例作进一步的介绍。

2.3.4　GPU-CPU 异构体系结构

CPU 与 GPU 一般经北桥通过 AGP 或 PCI-E 总线进行连接，各自拥有独立的外部存储器(分别为内存与显存)。在一些芯片组中，没有采用独立的显存芯片，而是使用了集成 GPU，它直接从内存中划分出一块区域作为显存。Intel 和 AMD 提出的 CPU-GPU 融合产品还准备直接将 CPU 和 GPU 通过快速通道互连(QPI)或高速串行总线(HT)进行连接，并集成在一块芯片内。

在 CPU-GPGPU 这样的异构体系结构中，GPGPU 作为 CPU 的协处理器完成图形计算和通用计算，GPGPU 以外部设备的形式通过 PCI-E 总线与 CPU 进行通信。CPU 和 GPU 各自拥有自己的存储系统，它们之间通过 DMA 操作实现数据的传递。作为主处理器的 CPU 采用目前广泛使用的单核或多核处理器，其体系结构不再介绍。

R600 包含的主要部件有数据并行阵列(DPPA)、命令处理器、片上内存单元、存储控制器。数据并行处理器阵列负责执行各种渲染和计算程序，命令处理器负责管理 GPGPU 的整个运行过程，片上内存单元是 GPGPU 的本地存储器件，存储控制器完成对系统主存和本地内存中数据的访问，如图 2-32 所示。

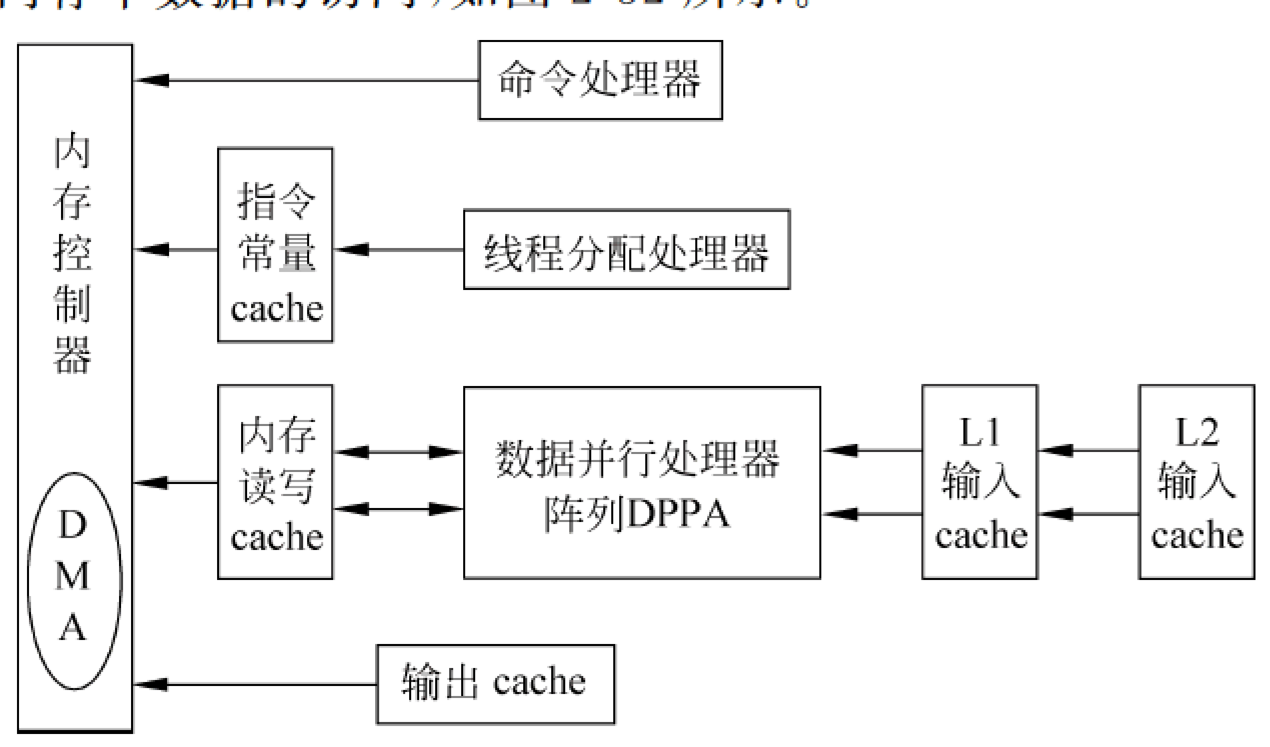


图 2-32　GPGPU 体系结构示意图

2.3.5 Fermi 架构

2010 年，NVIDIA 推出了代号为 Fermi 的新一代 CUDA 架构。Fermi 拥有超过 30 亿个晶体管、最多 512 个 CUDA Core，可实现超级计算特性与性能。与基于 CPU 的传统服务器相比，其成本仅为 1/10，功耗仅为 1/20。Fermi 的硬件结构如图 2-33 所示。

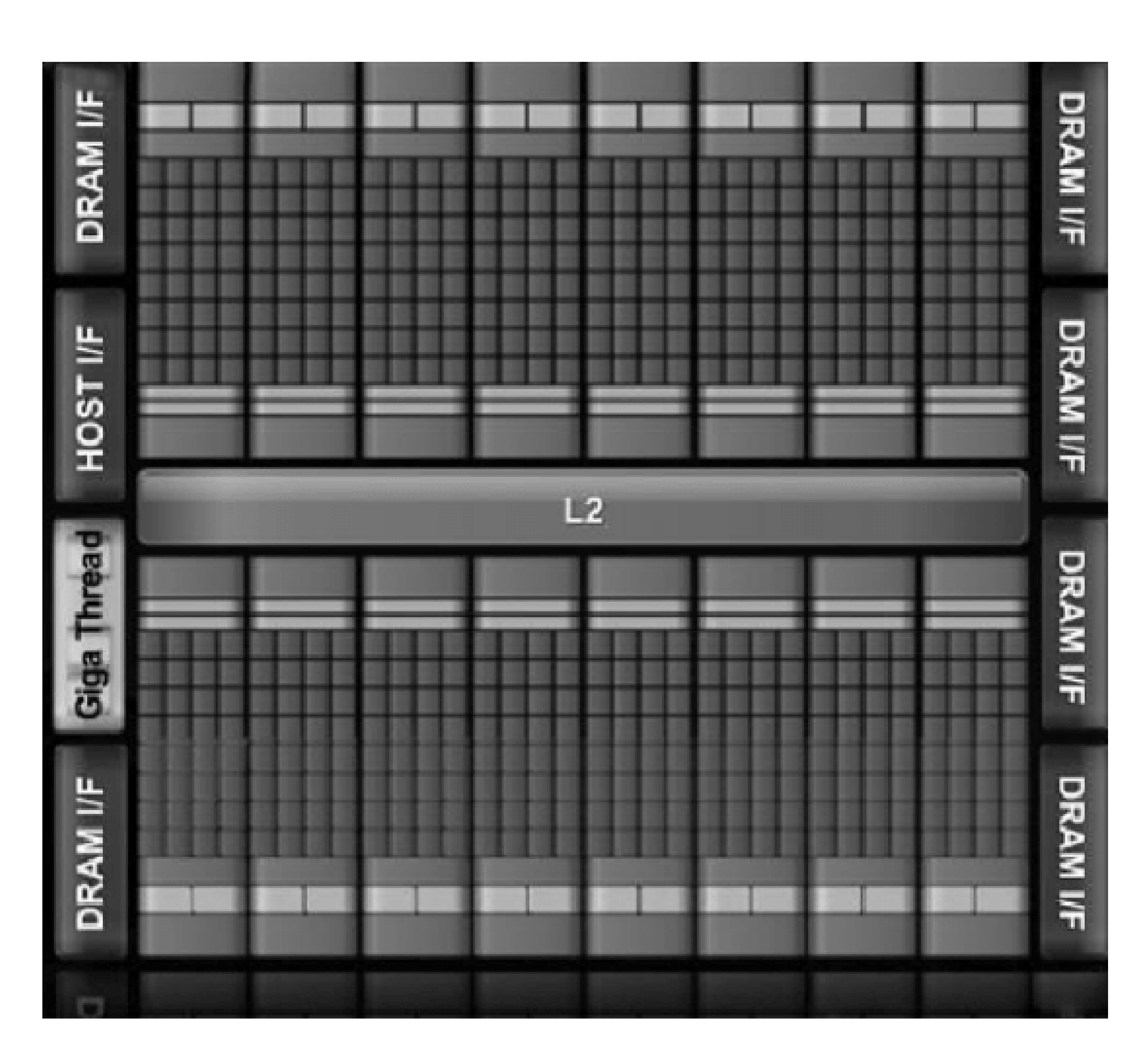


图 2-33　Fermi 硬件架构

Fermi 架构大大提高了 GPU 通用计算的实用性，它具有如下特点：

- 更多、更标准化的流处理器。在 G80/GT200 中，都是由 8 个流处理器构成一组流多处理器(SM)，而 Fermi 增加到由 32 个流处理器构成一组流多处理器，流处理器的数目最多为 16 组(少于 GT200 的 30 组)，但流处理器的总量则从 240 个增至 512 个，多于 GT200 的 240 个，是 G80 的 4 倍。所有流处理器现在都符合 IEEE 754-2008 浮点算法和完整的 32 位整数算法(在过去，仅能模拟 32 位整数算法，仅能计算 24 位整数乘法)。同时，引入的还有熔加运算(Fused Multiply Add/FMA)，每次循环操作单精度数 512 个、双精度数 256 个。符合业界标准，计算结果不会产生意外偏差。
- 双精度浮点的性能得到大大提高。峰值计算速度可以达到单精度浮点的 1/2，而在过去只有 1/8，而 AMD 现在也不过 1/5(比如 AMD 的 Radeon HD5870 的单精度峰值计算速度为 2.72TFLOPS、双精度峰值计算速度为 544GFLOPS)。
- ECC(ERROR CHECK AND CORRECT)支持。AMD Cypress 可以检测内存总

线上的错误，却不能修正。但是，NVIDIA Fermi 的寄存器文件、一级缓存、二级缓存、DRAM 均支持 ECC 错误校验。

- 统一的 64 位内存寻址。在以前的架构里，各种不同的载入指令，取决于内存类型如本地、共享、全局内存分别使用不同的指令。这就给使用带来了麻烦。而 NVIDIA Fermi 提供统一的地址空间，内存的地址取决于存储的位置：最低位是本地地址，然后是共享地址，剩下的是全局地址。
- 每组 SM 都有 16KB 的共享内存。这些共享内存由其中的 8 个 SP 使用，注意它们不是缓存，而是由软件管理的内存，可以写入/读取数据。为了满足应用程序和通用计算的需要，NVIDIA Fermi 引入了真正的缓存，每组 SM 拥有容量为 64KB 的可配置内存，可分为 16KB 的共享内存与 48KB 的一级缓存或者 48KB 的共享内存与 16KB 的一级缓存，能够灵活地满足不同类型程序的需要。NVIDIA Fermi 的整个芯片拥有一个容量为 768KB 的共享二级缓存，执行原子内存操作比 GT200 快 5～20 倍。
- 更短的程序切换开销。传统 GPU 内核在工作的时候会忽略程序之间切换带来的开销，在大规模的数学计算中，这个开销是相当大的。NVIDIA Fermi 的出现改变了这种状况，并将程序切换的开销减少到约 20～25μm。这意味着 Fermi 在某些环境下会表现出更优秀的性能。
- 全新的 Debug 支持。在过去，当需要进行 Debug 的时候，GPU 的整个工作状态将被暂停，然后基于 CPU 的 Debuger 将读取 GPU 的寄存器状态、线程状态和显存内容，完成 Debug 之后再恢复 GPU 的工作状态。目前，NVIDIA Fermi 的 Debug 不再需要 CPU 的介入，所有工作将由 GPU Trap Handler 软件来实现，GPU Trap Handler 还可处理 CPU 代码。
- 新的指令集架构（ISA）。NVIDIA Fermi 的指令集架构被大大扩充，它支持 DX11、OpenCL、C++、Visual Studio 等，当然也支持 C、Fortran 与 OpenGL 3.1/3.2。
- SM 的变化。SM 的执行不再以 half-warp 为单位：传统的线程模式中，Kernel 的执行是以 warp 为单位，每个 warp 包含 32 个 thread，这些 thread 分成两组，每次执行一组，也就是 half-warp。在 Fermi 架构中，在每个 SM 前端均有 2 个 warp 调度器和 2 个独立分配单元，它们与 SM 的其他部分完全独立，均可在 1 个时钟周期内选择发送一半 warp，而且这些线程可来自于不同的 warp。分配单元和执行硬件之间有一个完整的交叉开关（Crossbar），每个单元均可向 SM 内的任意单元分配线程。从 G80 开始，SM 单元被提出，到 GF100 时已发展到了第三代，此时每个 SM 均有 32 个 SP，为 G80/G92/GT200 的 SM 中处理器数量的 4 倍，如图 2-34 所示。

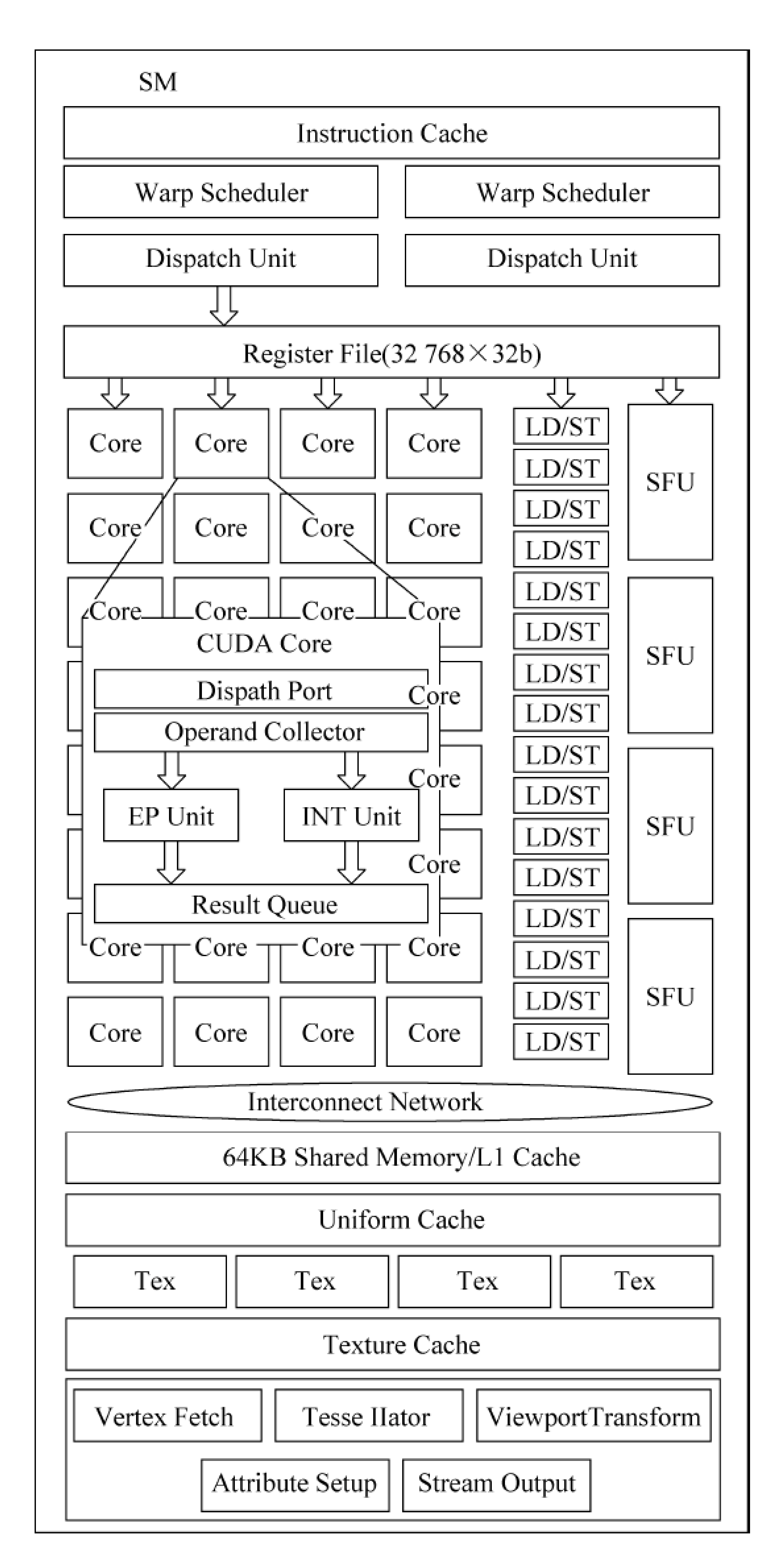


图 2-34 NVIDIA GF100 架构的 SM 单元

2.3.6　GPU 集群

1. GPU 集群概述及发展

与传统 CPU 集群一样,GPU 同样可以用来搭建集群以提高计算性能。从早期的 GPU 加速工作站到异构型 CPU/GPU 集群,再到现在基于单块集成电路的 CPU-GPU 服务器,GPU 集群已经开始在高性能计算领域发挥着重要的科研及商业作用。相对于传统的 CPU 集群架构,GPU 集群架构可提供成本更低、体积和功耗更小、性能更强的并行计算解决方案。中国首台千万亿次/秒超级计算机“天河一号”就采用了 NVIDIA Tesla M2050 组成了庞大的 GPU 集群,其处理内核总数超过 20 万颗,并在第 36 届超级计算机 Top500 排名中夺魁,成为当时世界上最快的计算机。

斯坦福大学的 Mike Houston 已经在 GPU 集群方面工作了一段时间,他所在的小组就正在尝试用并行 GPU 和 HMM(Hidden Markov Models 隐形马尔科夫模型)进行蛋白质研究。HMM 编码被重写,将其在 GPU 上运行,之后再被修正,数据库搜索被分成多个 GPU 集群结点。总体来讲,这并不是一个特别理想的集群,因为每个搜索都是独立的,因此全部搜索并非并行。但是每个结点的性能都很不错,大约达到了单一 CPU 的 10～40倍,更重要的是,并行编码扩展得非常好。

SUNY Stony Brook 的虚拟化实验室,在 GPU 分布式图像和分布式计算领域有着较深的研究。并在 2004 年发布了 Lattice Boltzmann Method(LBM)GPU 集群的研究报告,该报告是纽约时代广场的空气污染模拟状况。这个小组在研究的时候,采用了 GPU 作为集群中的结点来重写 LBM 编码。因为 GPU 不能直接访问网络界面,数据在传输时转换成 CPU 能识别的编码,而在结点处则被转换回 GPU 能够识别的编码。通过这种方法,程序员得到了 4.6 倍于 CPU 的速度提升。

2. GPU 集群硬件架构与组织方式

与单 GPU 环境相比,多 GPU 环境下程序编写复杂度将会提高。需要考虑到当前的 GPU 的能力、GPU 程序的资源占用情况、GPU 程序需要与 CPU 交互的数据量等。在通信方面则需要考虑任务分配、数据交换、使用冗余计算代替数据传输等问题。随着 NVIDIA 公司陆续发布新款服务器与 GPU 超级计算机,多 GPU 的并行执行技术日趋重要。与传统 CPU 程序的并行执行不同,GPU 程序的并行执行可被分为三级:

- 第一级是 GPU 内部的并行执行,也就是前面讨论的单 GPU 程序;
- 第二级是 GPU 与 CPU 的并行执行和协作;
- 第三级则是多 GPU 之间的并行执行。

GPU 集群可以有两种组建方式:

- 在一台计算机内，1 个 CPU 带多个 GPU；
- 在多台计算机内，每台计算机都是 1 个 CPU 对应 1 个或多个 GPU。

GPU 之间的通信只能通过所连接的 CPU。第一种方式实际上是第二种方式的特例，GPU 自己只能同本机的 CPU 通信，而不能同网络上其他 GPU 通信。NVIDIA 的 SLI 技术就是基于第一种方式的 GPU 集群。不过 SLI 技术本身只能支持图形运算，而且还是 Windows 平台特有的。位于不同 CPU 上的 GPU 之间的通信问题目前还是 GPU 集群方面一个正在探索的问题。GPU 集群的连接方式如图 2-35 所示。

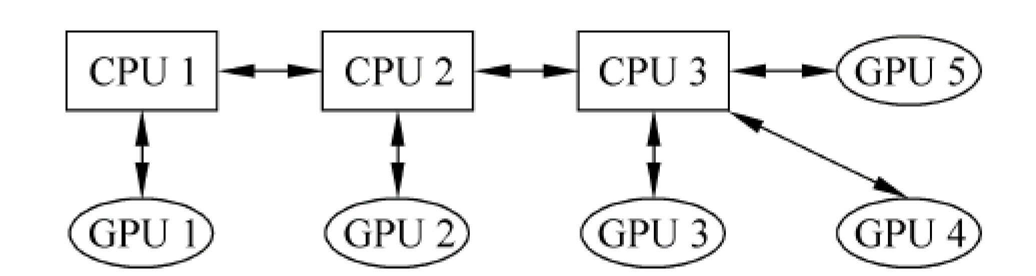


图 2-35 GPU 集群的组建方式



GPU 本身就是一个高度并行化的设备，它的执行方式类似于用刷子刷墙，要刷出纯色的墙效率会非常高，但较难刷出多变的图案，这点读者可以体会一下。本节以 NVIDIA 公司的 GPU 系列产品为主要内容，从架构方面介绍了 GPU/GPGPU 的发展、新型的 GPU 体系结构和 GPU 集群的基本概念，并简单介绍了 GPU 集群的相关知识。在阅读本节后应该对 GPU 的功能、结构特性有一定的了解。

2.4 Cell BE

Cell BE（Cell Broadband Engine）处理器是基于 CBEA（Cell Broadband Engine Architecture）架构的首个多核体系结构。Cell BE 处理器除了主要应用于索尼的 Play Station 3(PS3)游戏主机上，还应用在 3D 绘图、影音多媒体与科学运算等方面。

在 2000 年，Sony、Toshiba 和 IBM 三家公司开始探讨开发下一代游戏机。IBM 主要进行微处理器开发，Toshiba 作为大批量生产与开发的技术伙伴，Sony 作为内容提供商。Cell BE 处理器的设计目标是：

- 出色的性能，尤其在游戏/多媒体应用方面。最终性能达到 Play Station 2(PS2)处理器性能的 100 倍，在将来处于领导地位。
- 对用户和网络的实时响应。
- 适用于广泛的平台。

在经过几个月对架构的讨论之后，Sony、Toshiba 和 IBM 在 2001 年 3 月宣布正式成

立 STI(SCEI-Toshiba-IBM)设计中心。历时 5 年,投资额超过 4 亿美元,在 600 多人的研发团队的努力下,终于在 2005 年完成了基于 CBEA 架构的首个多核处理器芯片 Cell BE,如图 2-36 所示。

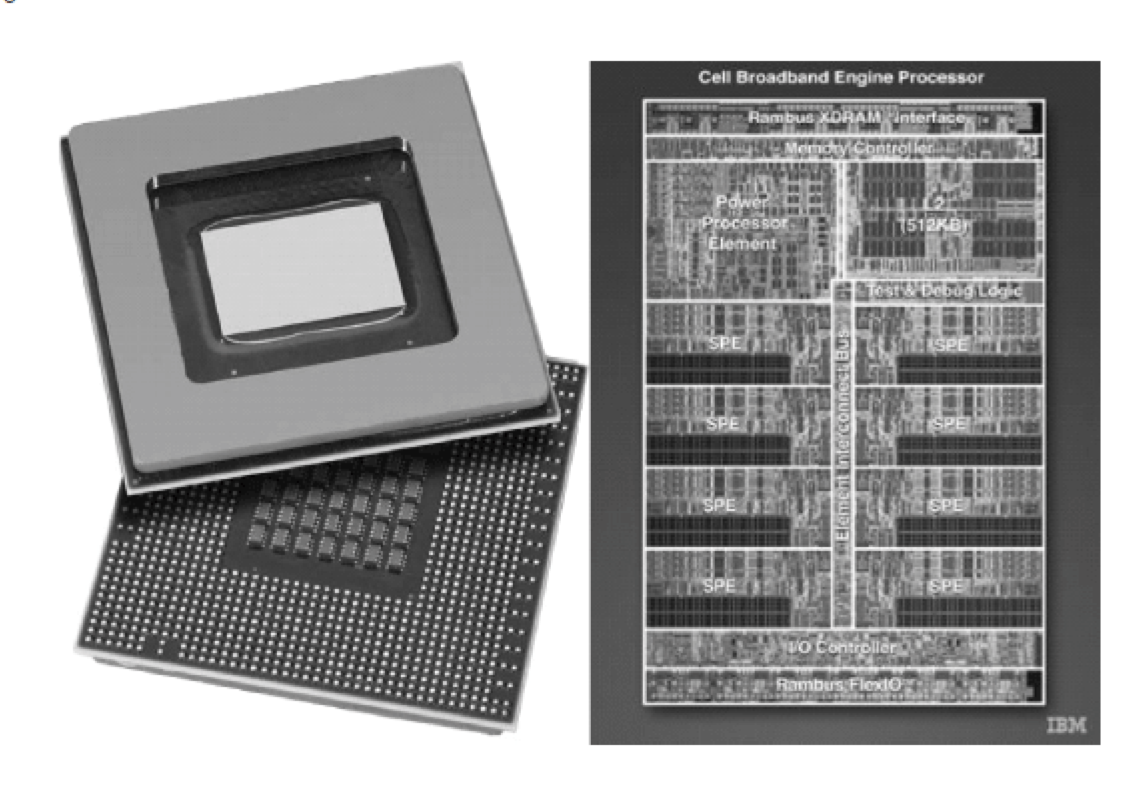


图 2-36　Cell BE 处理器芯片和硅片

Cell BE 处理器的成功实现得益于以下因素:

- STI 设计中心使用了综合的设计方法,包括处理器架构、硬件实现、系统结构和软件编程模型等;
- STI 设计中心对参加研制的各地科研机构发挥了非常关键的组织领导作用;
- Cell BE 包含了许多灵活的设计(包括可重新编程的协处理器和可重新配置的I/O 接口),使得 Cell BE 成为支持多系统的高效芯片。

Cell BE 处理器的研制成功,丰富了计算机体系结构,为异构多核处理器的发展奠定了基础。

2.4.1　Cell BE 概述

Cell BE 处理器是一种典型的片上异构多核处理器,它包括一个主处理单元(PowerPC Processing Element, PPE)和 8 个协处理单元(Synergistic Processing Element,SPE),接口部分包括一个内存控制器(Memory Interface Controller,MIC)和一个宽带引擎接口(Broadband Engine Interface, BEI),这些部件通过单元互连总线(Element Interconnect Bus,EIB)连接起来。Cell BE 处理器的系统结构图如图 2-37 所示。

1. PowerPC 处理器

PowerPC 处理器(PowerPC Processor Element,PPE),是一个通用的双线程 64 位 RISC 处理器,其设计遵循 PowerPC 架构 2.02 版本,并进行了 Vector/SIMD 多媒体指令

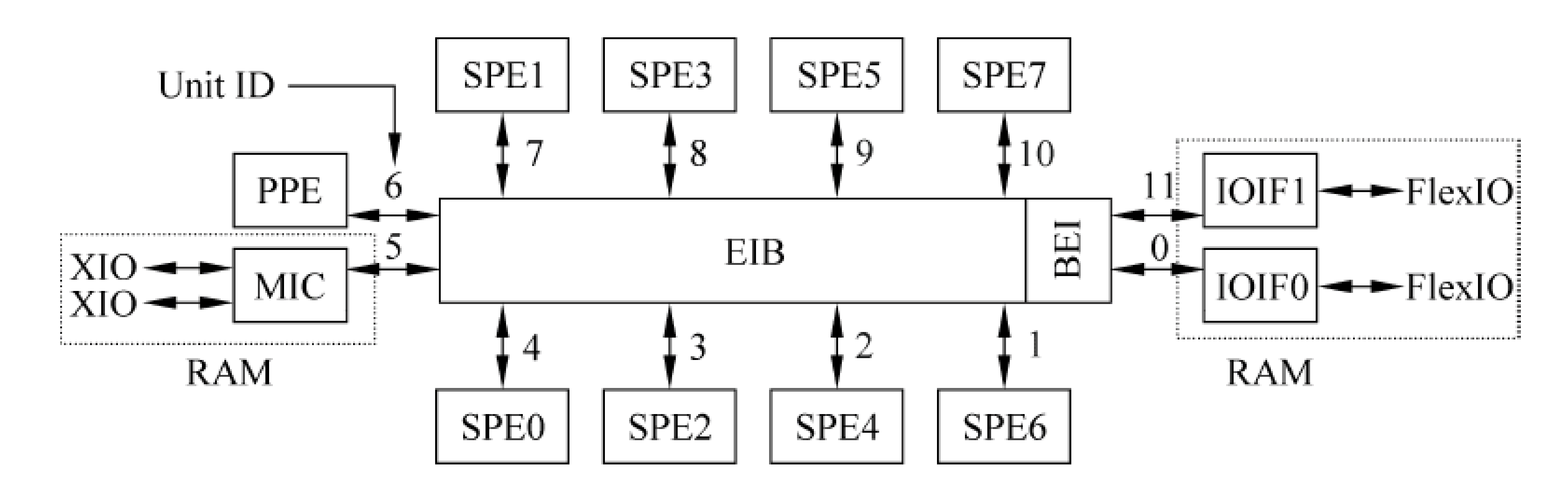


图 2-37 Cell BE 处理器的系统结构图

扩展。比如，为 PowerPC970 处理器编写的程序，无须进行任何的更改就可在 Cell BE 上运行。PPE 负责整个 Cell BE 系统的全面控制，并且为运行在 PPE 和 SPE 上的应用程序提供操作系统平台。PPE 包含两个主要单元：PowerPC 处理器单元(PowerPC Processor Unit，PPU)和 PowerPC 处理器存储子系统(PowerPC Processor Storage Subsystem，PPSS)，如图 2-38 所示。

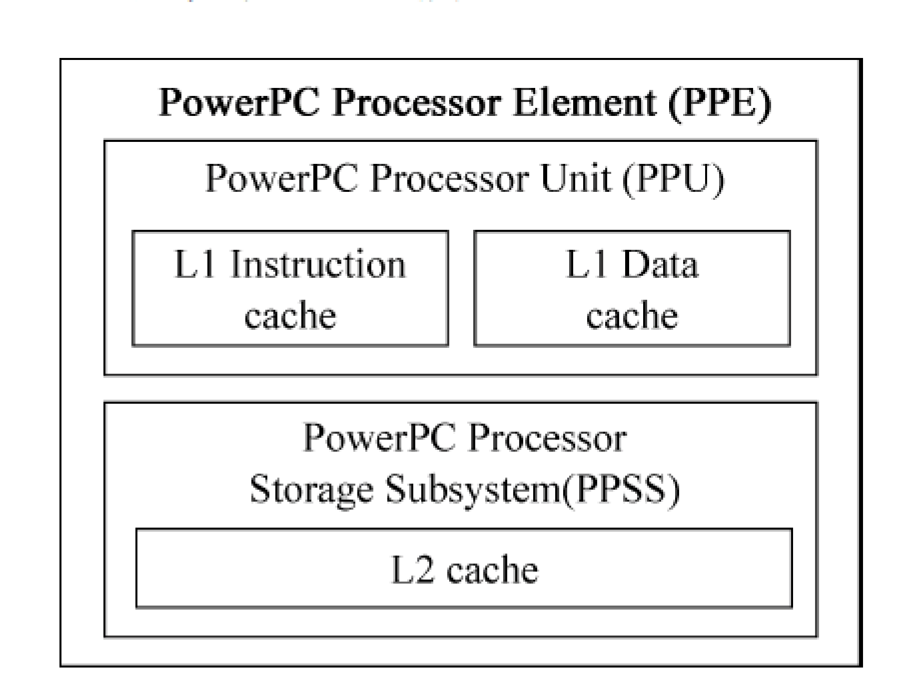


图 2-38 PPE 的系统结构图

PPU 执行 PowerPC 架构指令集和 Vector/SIMD 多媒体扩展指令集，它有一个容量为 32KB 的一级指令缓存、一个容量为 32KB 的一级数据缓存和六个执行单元。PPU 能够在每个时钟周期内独立地载入 32B，存储 16B，并且保证内存一致性。它支持两个并发线程，在每个时钟周期内能完成两个双精度操作，峰值速度在主频 3.2GHz 时达到 6.4GFLOPS。

PPSS 处理所有的 PPU 内存访问和 EIB 的内存一致性操作，它有一个容量为 512KB、8 路组相连、带有校验码的写回二级缓存。像一级缓存一样，二级缓存的行大小也是 128B。PPSS 的二级缓存有一个主存的读写接口，支持 8 个软件管理的数据预取流。它包含一级数据缓存的内容，但是不保证包含一级指令缓存的内容，它支持对称多处理器(Symmetric Multiprocessor，SMP)的完全一致性。

PPSS 与 EIB 之间的接口支持 16B 大小的载入存储总线，在同一时刻只能进行一种存储访问，所有的内存访问按照程序中设置的顺序进行。该接口支持资源分配管理(Resource Allocation Management，RAM)，允许享有特权的软件来控制各种资源分配的时间段。二级缓存和旁路转换缓冲区(Translation Lookaside Buffer，TLB 或称页表缓冲区)使用了替换管理表(Replacement Management Table，RMT)，替换管理表允许享有特权的软件对二级缓存和 TLB 的使用进行控制，这在实时编程时尤其有用。

PPE 的主要功能是控制执行、运行操作系统以及管理系统资源和 SPE 的线程，可以

运行现有 PowerPC 架构的软件，而且非常适合执行系统控制代码。PPE 的指令集是扩展的 PowerPC 指令集，包含 Vector/SIMD 多媒体扩展指令集，可进行两种模式的运算(64 位和 32 位模式)，运算的模式控制了地址解析的方式以及状态位的设置。

2. 协处理器

Cell BE 中的协处理器(Synergistic Processor Element，SPE)是基于 RISC 的 128 位处理器，适用于计算密集、数据密集的 SIMD 与标量处理等方面的应用。它主要由协处理单元(Synergistic Processor Unit，SPU)和内存流控制器(Memory Flow Controller，MFC)组成，其系统结构如图 2-39 所示。为了降低功耗，需要优化计算密集型和多媒体应用的性能，为此协处理器单元(SPU)执行了一套新的 SIMD 指令集。

每个 SPU 都有一个 256KB 大小的本地存储(属于 SPU 的私有存储空间)，它并不与系统内存进行统一编址。SPU 从 256KB 的本地存储器中取得指令，可在本地存储器与寄存器文件之间载入和存储各种类型的数据。其中，寄存器文件有 128 个寄存器，每个 128 位宽。每个 SPU 有四个执行单元、一个 DMA 接口、一个与 MFC、PPE 和其他设备进行通信的通道接口。

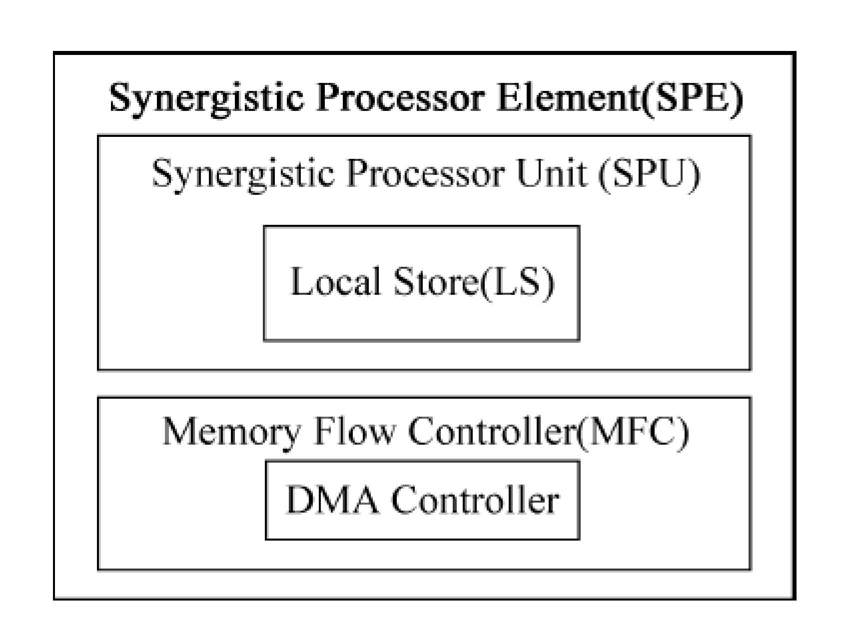


图 2-39 SPE 的系统结构图

每个 SPU 都是一个独立的、拥有自己的程序计数器、优化运行程序的处理器单元。SPU 通过 MFC 进行 DMA 数据传送，把程序和数据存入到本地存储中。MFC 使用 DMA 控制器实现 DMA 数据传送。这样，SPU 就从它的本地存储器取指令并执行，同时也在本地存储器中进行数据的载入和存储。

本地存储器(Local Store，LS)是一个 256KB 大小、ECC(Error Checking and Correcting)保护、单端口、没有缓存的存储器，它存储 SPU 使用的所有指令和数据。本地存储器每周期支持一次来自其他 SPE 程序的访问或 DMA 数据传送访问。SPU 在每个周期内可预取指令 128B，数据访问宽度是 16B，DMA 访问宽度为 128B。SPU 使用载入和存储指令访问它的本地存储，不用进行地址变换。系统中，本地存储器与主存之间的 DMA 传送满足一致性要求。PPE 能够将其创建的数据结构的指针通过主存空间传送到一个 SPU 上，SPU 使用该指针来执行一个 DMA 命令，将对应的数据结构传回本地存储器。映射到内存的邮箱或原子 MFC 同步命令可用来进行同步或互斥操作。

每个 SPU 都拥有自己的 MFC。MFC 是 SPU 的接口，通过 EIB 与主存、其他处理器单元和系统设备进行通信。MFC 的主要功能是提供本地存储和主存之间的接口，它通过 DMA 控制器在本地存储器和主存之间移动指令和数据。MFC 还支持 DMA 数据传送的

主存端内存保护、主存和本地存储间的同步和与 PPE、其他 SPE 和设备之间的通信等功能。

SPE 没有缓存，在容量为 256KB 的本地存储器上进行操作，在本地存储器中存储了 SPE 所需的指令和数据。每个 SPE 都包含有 MFC，由 MFC 使用异步一致性 DMA 操作来完成数据和指令在本地存储器与系统存储器之间的传输。进行 DMA 操作的编程方式有如下三种：

(1) 在 SPE 上使用指令将 DMA 命令插入到处理队列中；

(2) 在本地存储中，准备 DMA 传输命令列表，发出"DMA 列表"命令；

(3) 在系统中其他处理器上，使用存储或 DMA 写命令，把 DMA 传输命令插入处理队列中。

3. 单元互连总线

单元互连总线(Element Interconnect Bus，EIB)是 Cell BE 处理器上所有单元、片上存储控制器以及 I/O 的数据和命令通信渠道，提供的峰值带宽高达 204.8GB/s，如图 2-40 所示。它支持内存一致性和对称多处理器(Symmetric Multiprocessor，SMP)操作，可以将多个 Cell BE 处理器互连组成一个多处理器集群系统。

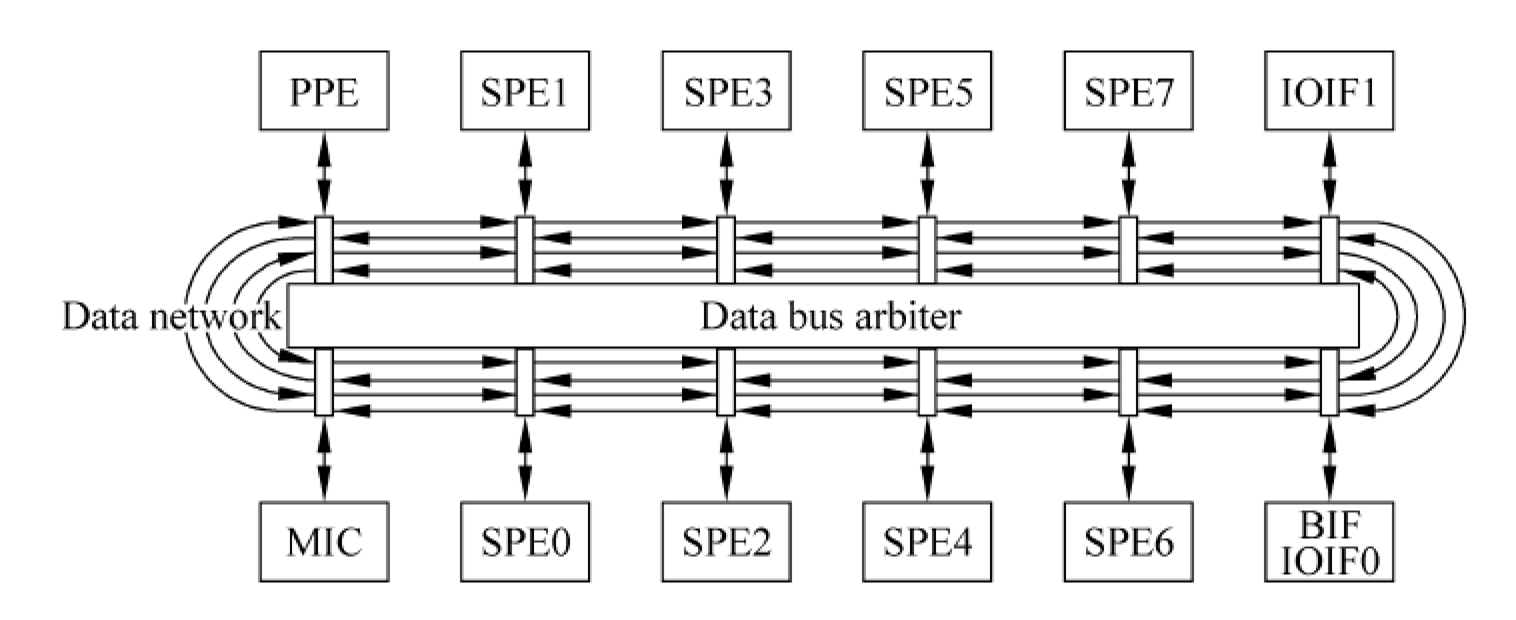


图 2-40 Cell BE 处理器内部元件之间的互连数据总线拓扑结构图

EIB 由 4 个 16B 宽的数据环组成，每个数据环一次可以传送 128B(即 PPE 缓存中的一行)，处理器单元可通过 EIB 同时发送和接收数据。图 2-37 给出了各个单元的 ID 号和各个单元连接到 EIB 的顺序。对程序员来说，各个单元连接到 EIB 的顺序对获得 EIB 上的最小传输延迟非常重要：传输延迟与连接顺序的间隔数密切相关，所以邻近的单元之间的延迟最小，间隔为 6 的单元间的延迟最大。

EIB 内部的最大带宽是每个处理器时钟周期内为 96B，在每个数据环上可同时进行多路数据传送，包括主存与 SPE 之间超过 100 个的 DMA 内存请求。EIB 除了保证传输的进行，并不支持任何 QoS 服务，然而享有特权的软件可使用 EIB 中的资源管理部件(Resource Allocation Management，RAM)来控制资源请求者(PPE、SPEs 和 I/O 设备)

对内存和 I/O 资源的使用率。

4. 内存控制器

Cell BE 芯片上的内存接口控制器(Memory Interface Controller,MIC)提供了 EIB 与物理内存之间的接口。Cell BE 支持一个或两个 XDR(Extreme Data Rate,XDR) Rambus 内存接口,使用两个 XDR 接口能支持 64MB～64GB 的 XDR DRAM。

在每个 XDR 接口上,内存访问的大小可从 1 到 8、16、32、64、128B,队列中最多可排有 64 个读操作和 64 个写操作,资源分配信号管理器可以提供队列中排队的操作的数量。

MIC 有多种软件控制模式,包括快速路径模式(当命令队列为空时,用于降低延迟)、高优先级读模式(在所有读操作中,SPE 的读操作优先)、提前读模式(在前一个写操作完成之前,开始读操作)、猜测读模式和慢模式(用于能耗管理)等。

XDR DRAM 内存是 ECC 保护的,带有多位错误检测和位选择错误修正功能,支持写屏蔽,支持初始时间校准和周期性时间校准,支持动态宽度控制、子页面激活、动态时钟门控和 4、8 或 16 个内存通道。

5. 宽带引擎接口

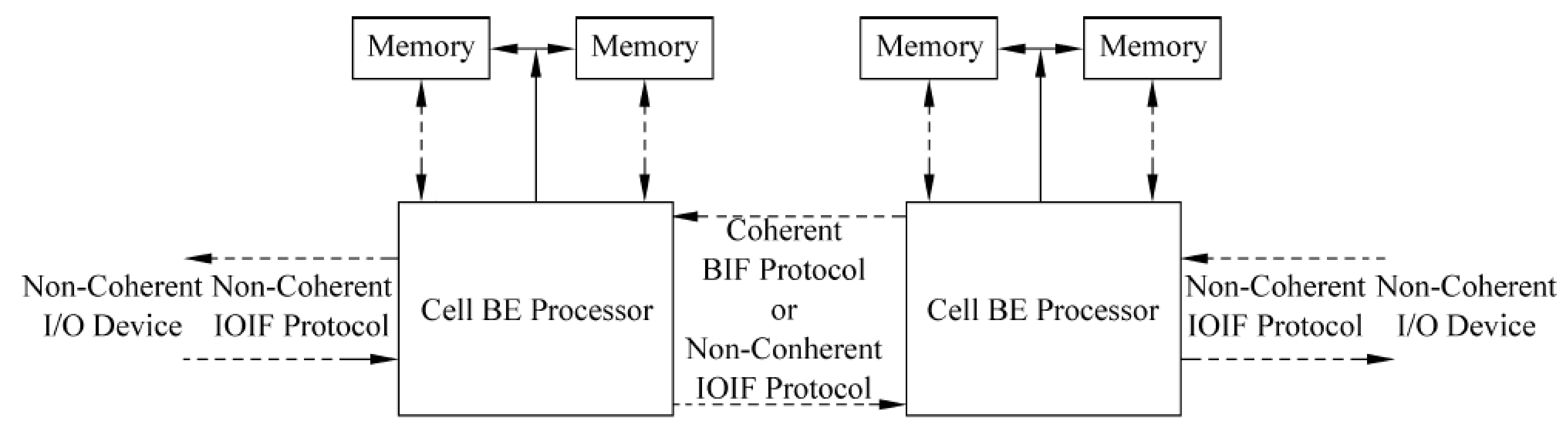
Cell BE 芯片上的宽带引擎接口(Broadband Engine Interface,BEI)单元支持 I/O 接口,它包括一个总线接口控制器(Bus Interface Controller,BIC)、I/O 控制器(I/O Controller,IOC)和内部中断控制器(Internal Interrupt Controller,IIC),管理着 EIB 与 I/O 设备之间的数据传送,提供 I/O 地址变换和命令处理。

BEI 支持两个 Rambus FlexIO 接口:IOIF0 和 IOIF1。其中,IOIF1 只支持非耦合的 I/O 接口协议,适合连接 I/O 设备。IOIF0(也称为 BIF/IOIF0)可通过软件配置选择非耦合 IOIF 协议或内存耦合 Cell BE 接口(Cell Broadband Engine Interface,BIF)协议。BIF 协议是 EIB 的内部协议,可通过 IOIF0 连接到另外一个 Cell BE 处理器。图 2-41(a)与图 2-41(b)分别是两个 Cell BE 处理器之间与四个 Cell BE 处理器之间的互连结构图。

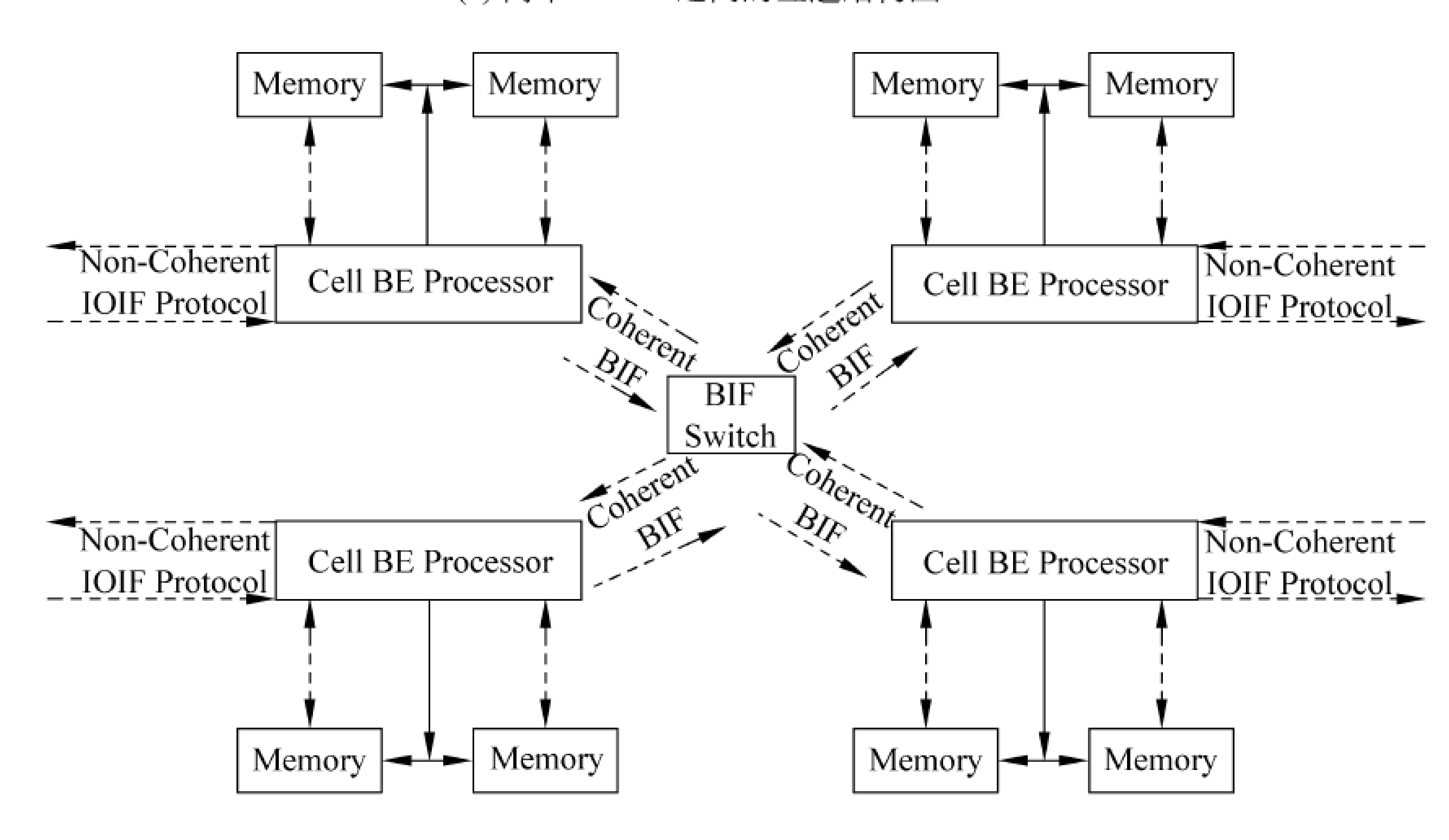
2.4.2 Cell BE 关键技术

1. cache 管理

在 Cell BE 处理器中,有几种类型的 cache。其中,首先是 PPE 的一级指令缓存和一级数据缓存以及与一级缓存保持一致性的二级缓存。这些缓存中的内容与主存的中内容保持一致,PowerPC 架构的缓存控制支持用户对 cache 行的操作。除了一级缓存和二级缓存,PPE 和 SPE 还有其他缓存、队列和数组用于提高性能,并可使用软件方法来进行控制。



(a) 两个Cell BE之间的互连结构图



(b) 四个Cell BE之间的互连结构图

图 2-41 Cell BE 之间的互连结构图

1）PPE caches

除了一级缓存和二级缓存之外，PPE 还有其他缓存、队列和数组来支持内存管理，作为一级缓存和二级缓存的前驱（predecessors）和扩展（extensions）。比如，PPE 有存储队列，可以存放载入单元、存储单元和缓存之间传输的数据；PPE 有移出（castout）队列，可以存放移出二级缓存已被修改的数据。PPE 的线程共享所有的一级缓存、二级缓存和一些其他存储结构（在有的存储结构中，每个线程都拥有一份副本）。

两个 PPE 线程共享执行单元、微代码（microcode）引擎、指令预取控制、PPSS、一些其他缓存、数组、队列和存储结构。

PPU 支持缓存、数组、队列和其他存储结构，其主要模块有指令单元（Instruction Unit，IU）、载入与存储单元（Load and Store Unit，LSU）、内存管理单元（Memory Management

Unit，MMU）。PPSS 的主要模块包括内核接口单元（Core Interface Unit，CIU）、非缓存单元（Noncacheable Unit，NCU）和二级缓存。CIU 包括载入、存储和再载入子单元，它们之间独立地进行工作。

（1）L1 cache

PPE 具有支持 PowerPC 架构缓存控制指令的一级指令缓存和一级数据缓存。通过设置页表项（Page-table Entry，PTE）的缓存控制位来确定是否对主存的访问（载入、存储和指令预取）进行缓存。如果进行缓存的话，则缓存控制位必须设置为'0'。

PPE 的两个线程动态地共享一级指令缓存和一级数据缓存，其中一个线程载入的缓存模块也可被另一个线程使用。一级指令缓存是 PPE 指令单元的主要部分，一级数据缓存是 PPE 载入/存储单元的主要部分。处理单元以 Cell BE 的最高时钟频率对一级缓存进行访问。

一级缓存支持 PowerPC 架构的缓存管理指令，完成如写回、便无效、刷新（写回和无效）、将目录清为'0'等操作。

一级指令缓存使用完全二叉树最近最少使用替换算法（true binary Least Recently Used，LRU），不带替换管理表（Replacement Management Table，RMT）锁定；一级数据缓存使用伪最近最少使用替换策略（pseudo Least Recently Used，p-LRU）。

（2）L2 cache

PPE 包含一个容量为 512KB 的二级缓存，该二级缓存支持 PowerPC 架构缓存控制指令，通过 PTE 中的缓存控制位来控制缓存的使用。

二级缓存保证系统内的缓存行的全一致性，能为其他处理器单元提供数据。从逻辑上讲，二级缓存是内联缓存，它是采用写回方式来保证缓存的一致性，包含一级数据缓存的全部目录，但不保证包含一级指令缓存的目录。两个 PPE 执行线程动态地共享二级缓存，一个缓存块可被其中一个线程载入，被另一个线程使用。

二级缓存处理所有可缓存的载入和存储、数据预取、指令预取、缓存操作和栅栏操作。二级缓存像一级缓存一样支持缓存管理指令。

二级缓存可通过 API 函数设置如下替换算法：

- 二叉树 LRU 模式，它是一种基于二叉树的调度算法；
- 直接映射模式，它使用 3 个标签地址位将 1 个地址映射到 8 个同余类成员中；
- 伪 LRU 模式，它使用软件配置的地址范围寄存器（Address Range Register，ARR）和一个替换管理表锁住到某个指定的替换类别标识符（Replacement Class Idetifier，RclassID）的缓存通道，其他两种模式不能使用这个功能。

2）SPE caches

每个 SPE 的 MFC 均有如下两种缓存（PPE 特权软件可对它们进行管理）：

- 旁路转换缓冲区（Translation Lookaside Buffer，TLB）；

- 原子缓存(Atomic Unit Cache,called the Atomic cache)。

(1) 旁路转换缓冲区

在 MFC 中,每个 SPE 的协内存管理单元(Synergistic Memory Management,SMM)包含:一个 256 项(entry)4 路组相连的 TLB 缓存,缓存中存有最近使用过的页表项(Page Table Entry,PTE)。TLB 有 64 个同余类(congruence classes),每个同余类需要用 6 位进行 TLB 索引。TLB 是奇偶校验保护的,奇偶校验的产生和检查可通过寄存器设置实现。可使用硬件和软件的方法替换 TLB 行。

(2) 原子缓存

每个 SPE 的 MFC 都包含一个原子单元,处理 SPU 的信号操作,为 SMM 提供 PTE 数据。原子单元还有如下功能:

- 为 MFC 原子命令提供原子操作;
- 为 SMM 的硬件查找表提供 PTE,更新 PTE 中的引用位或变化位;
- 通过监听操作,保持缓存一致性。

原子缓存存储 6 组 128 字节缓存行大小的数据。其中,4 组支持信号操作,1 组支持 PTE 访问,1 组支持数据的重新载入。

3) 替换管理

PPE 和 SPE 为管理软件提供了一种通过替换类别标识(Replacement class ID)控制二级缓存和 TLB 缓存的替换方法。其中,ID 用作 RMT 的索引,可锁住二级缓存和 TLB 缓存的行。RMT 的锁功能改变了二级缓存和 TLB 伪 LRU 算法的操作方式。

PPE 通过 RMT 管理它的 TLB 和它的二级缓存,每个 SPE 也有一个 RMT 来管理它的 TLB。当一小部分页面被应用程序频繁访问且需要在二级缓存或 TLB 中上锁以免缺失时,RMT 非常有用。比如,在实时应用程序中经常使用这种方法,但是,过多的资源锁操作对性能有不利影响。

2. 片内通信

Cell BE 作为一种片上多核架构,具有共享内存系统的许多属性,PPE 和所有 SPE 可保证一致性访问主存。但是,Cell BE 处理器不是传统的共享内存多核处理器,比如,一个 SPE 可运行程序,并可直接从它私有的本地存储中载入和存储数据。在传统的共享内存多核处理器中,多核之间的数据通信和同步操作至少部分依赖于共享内存。而 SPE 没有共享内存,它们必须显式地与其他单元进行通信,主要使用如下三种通信方式:

- DMA 传送;
- 邮箱消息;
- 信号通知。

这三种通信方式(见表 2-3)由 SPE 的 MFC 来实现和控制。

表 2-3　Cell BE 通信方式

通信方式	描　述
DMA 传送	用于主存与本地存储之间数据和指令的传送。SPE 依靠异步 DMA 传送来隐藏内存访问延迟和 SPU 计算时产生的并行移动数据的传送开销
邮箱	用于控制 SPE 与 PPE 或其他单元之间的通信。邮箱中的信息为 32 位，每个 SPE 有两个邮箱用于发送信息，一个邮箱用于接收信息
信号通知	用于控制从 PPE 或其他单元发出的通信。信号通知使用 32 位寄存器，能够配置成一发一收或多发一收等信号通知方式

有的编程模型要依赖 PPE 完成应用程序的任务管理，将任务指定和分配给 SPE。任务管理中，非常重要的一个部分就是将程序和数据载入主存，然后通过邮箱或信号通知寄存器通知可进行工作的 SPE。SPE 得到消息或通知信号之后，进行一次 DMA 操作，将主存中的数据和代码传送到它的本地存储。对该编程模型进行改进，PPE 可进行 DMA 操作，当 DMA 操作完成之后发送一个消息或信号通知 SPE。

处理完数据之后，SPE 可再进行一次 DMA 操作将结果传送到主存。当 DMA 操作将数据结果从本地存储传送到主存之后，SPE 会将一个完成消息写入用于发送消息的一个邮箱中，通知 PPE 数据处理和传送已完成。如果完成消息的大小超过 32 位，则可通过 SPE 写入多个邮箱消息或使用一个 DMA 操作完成长消息到主存的传送。在主存中，PPE 能够读到该消息。甚至可用 DMA 操作传送一个长的完成消息，使用邮箱通知 PPE 有信息。

1）DMA 传送

按照传输方式的不同，DMA 传送(参考自文献[18])可分为 DMA 传输和 DMA 列表传输。

- DMA 传输方式是在本地存储器的连续区域与系统内存中的连续单一区域之间进行传输，每次传输的数据大小不超过 16KB；
- DMA 列表传输方式是在本地存储器的单个连续区域与系统内存中的不连续区域之间进行传输，它通过一系列传输列表元素项来指定传输参数。

DMA 传输方式和 DMA 列表传输方式的特性如表 2-4 所示。

(1) DMA 传输方式

使用 DMA 传输方式进行传输的主要步骤如下：

① 调用 DMA 传输 API，发送传输命令；

② 设置标签掩码，确定进行状态检查的标签组；

③ 等待 DMA 传输完成。可在进行状态检查的标签组中的所有 DMA 传输全部完成之后才返回，也可在任意一个 DMA 传输完成之后就返回。

DMA 命令用于在 SPE 的本地存储与系统内存之间传输数据。系统内存通过 DMA 命令中的有效地址操作数进行寻址，SPE 的本地存储通过本地存储地址操作数进行寻址。每次 DMA 传输数据在 16KB 之内，本地存储数据以最小步长 16B 顺序访问。

表 2-4 DMA 传输方式和 DMA 列表传输方式的特性

方　式	特　性
DMA 传输	• 传输大小只能是 1、2、4、8 和 16 的整数倍； • 每次 DMA 的最大数据量为 16KB； • 传输的源起始地址和目的地址只能从 16B 的整数倍地址开始； • 128 字节对齐能够获得更好的性能
DMA 列表传输	• 一条 DMA 列表命令最多可包含 2048 条传输请求； • 每条传输请求最多可传输 16KB 的数据； • 每条传输请求的大小只能是 1、2、4、8 和 16 的整数倍； • 传输的源起始地址和目的地址只能从 16B 的整数倍地址开始； • 实现收集/散播的功能

当传输的源起始地址和目的地址与缓存行边界对齐且至少传输一个缓存行大小时，DMA 数据的传输性能最佳。16B 对齐的数据传输除了第一次和最后一次，剩余部分数据的传输能产生完整缓存行的总线请求，而第一次和最后一次可能会导致少于 128B 的部分缓存行传输。

(2) DMA 列表传输方式

使用 DMA 列表传输方式进行传输的主要步骤如下：

① 填充 DMA 列表数据结构；

② 调用 DMA 列表传输 API，发送传输命令；

③ 设置标签掩码，确定要进行状态检查的标签组；

④ 等待 DMA 列表传输完成。可在进行状态检查的标签组中的所有 DMA 传输全部完成之后才返回，也可在任意一个 DMA 传输完成之后就返回。

DMA 列表包含多个传输表项，它和发起 DMA 列表的命令一起，指定在 SPE 本地存储器的连续区域与系统内存中可能不连续的区域之间进行一系列的 DMA 传输。DMA 列表命令只能由运行于 SPE 上的程序发起，但 PPE 或其他单元能在 SPE 的本地存储器中创建和存储列表。DMA 列表命令可用于系统内存与本地存储器之间的收集/散播。

2) 邮箱消息

邮箱支持在 SPE 与其他单元之间发送和缓冲 32 位消息，比如 SPE 与 PPE 之间或者一个 SPE 与和其他 SPE 之间。每个 SPE 能够访问 3 个邮箱通道，每个邮箱通道都与 SPU 中的 MFC 的寄存器相连。两个单数据项输出邮箱通道——SPE 写输出邮箱和 SPE 写输出中断邮箱，用于从 SPE 发送消息到 PPE 或其他单元。一个 4 数据项邮箱——SPU 读输入邮箱，用于从 PPE、其他 SPE 或单元发送消息到 SPE。每个输出邮箱通道都有一个相应的内存映射 I/O(Memory Mapped I/O，MMIO)寄存器，PPE 或其他单元可通过访问 MMIO 寄存器来访问输出邮箱。

(1) 邮箱的读写

当一个 SPE 程序使用一个 SPE 写通道命令向一个输出邮箱写数据时,其他任何处理器单元或设备均可通过读主存空间中相应的 MMIO 寄存器来获得该数据。一个设备使用 MMIO 寄存器将数据写入 SPU 输入邮箱,SPE 程序可使用读通道命令读取这个邮箱获得数据。

输出邮箱的 MMIO 读操作或输入邮箱的写操作,能够通过编程来产生一个 SPE 中断,从而按顺序引发一个 SPU 中断。每当在一个 PPE 程序中 SPU 读输入邮箱时,SPU 输入邮箱通道计数器就会增加计数。每当在一个 SPE 程序中读取输入邮箱中的数据时,该通道计数器就会减少计数。输入邮箱像先进先出队列那样工作,SPE 程序首先读取最早进入邮箱的数据。如果在 SPE 程序读取该数据之前,PPE 程序写数据超过了 4 次,则通道计数器将停留在'4',且在第 4 个位置上存放 PPE 最后一次写入的数据。比如,如果 PPE 程序在 SPE 程序读取数据之前写了 5 次数据,则数据读取的顺序是第 1、第 2、第 3 和第 5 次写入的数据,第 4 次写入的数据被覆盖。

(2) 邮箱阻塞

SPE 对邮箱的读写操作为阻塞式操作。如果输出邮箱已满,SPE 对输出邮箱的写操作将阻塞 SPE,直到邮箱中的数据项被 PPE 读取。读操作与写操作类似,如果邮箱中没有消息而进行了读操作——不是读取通道计数值,则将阻塞 SPE,直到 PPE 向该邮箱中写入消息。也就是说,如果通道计数器是'0',则该通道就是一个阻塞通道,对该通道的读或者写操作将阻塞 SPE,直到通道计数器的值从'0'变为非零值。

为了避免阻塞,SPE 软件在决定是否读或写邮箱通道之前,要读取与邮箱相关联的通道计数器来避免阻塞 SPE。SPE 的阻塞不适用于 PPE,如果 PPE 发送消息到输入邮箱,且邮箱已满,PPE 也不会被阻塞。

当写通道指令要将数据发送到一个已满的输出邮箱时,写通道指令将被阻塞,SPE 程序不会覆盖输出邮箱中的数据。输出邮箱中的所有消息必须被 SPU 之外的单元读取,这样才有空间提供给更新的消息使用。相比较而言,其他设备对 SPU 的输入邮箱的写操作不被阻塞,因为输入邮箱的数据可被覆盖。当 PPE 或其他设备向一个已满的输入邮箱中写入消息时,之前最近一次写入邮箱的信息就会被覆盖而丢失。

(3) 邮箱的使用

使用邮箱进行消息通信的最大长度为 32 位,如缓冲完成标志或程序状态。邮箱能够传送任意短的数据,如发送存储地址、函数参数、命令参数和机器状态参数等。

当 SPE 将计算结果通过 DMA 方式存入主存时,邮箱非常有用。SPE 在请求 DMA 传送之后,等待 DMA 传送完成,然后通过写输出邮箱通知 PPE 计算已完成。

如果在等待一个 DMA 传送完成之后,SPE 才发送一个邮箱信息,这只能确保它的本地存储缓冲区能被重新使用,不能保持数据已保持一致性地写入了主存。SPE 通过在通

知 PPE 之前运行一个 MFC 同步命令解决该问题。但这个方法的效率不高，更好的办法是使 PPE 接收信号通知，然后在访问任意一个结果数据之前运行同步载入命令。

另外一种方法是，SPE 可通过 DMA 方式写一个通知给主存用来通知 PPE 它已完成计算，PPE 能够读取该通知。在这种方式中，数据和写回必须是顺序的。为了保证顺序，必须在数据 DMA 命令与主存通知之间运行强制顺序执行命令。

尽管邮箱主要用于 PPE 与 SPE 之间的通信，它们也能被用于一个 SPE 与其他 SPE、处理器或设备之间的通信。为了实现这一点，需特权软件允许一个 SPE 通过把目标 SPE 的状态区域映射到源 SPE 的有效地址空间访问该 SPE 中的邮箱寄存器。如果软件不能实现这一点，则只有原子操作和信号通知能够实现 SPE 与其他 SPE 之间的通信。

3）信号通知

SPE 的信号通知通道(Signal-notification channel)与 SPE 的输入寄存器相连，PPE、其他 SPE 和设备使用信号通知寄存器将信息(比如，缓冲区已满的同步标志)发送给一个 SPE。一个 SPE 有两个 32 位信号通知寄存器，每个寄存器有一个相对应的 MMIO 寄存器，这个 MMIO 寄存器可以写入信号通知数据。

当 SPE 程序读取一个信号通知通道时，硬件会自动清除通道。相比之下，主存空间中的 MMIO 读操作不会清除信号通知寄存器。在等待一个信号通知出现时，SPE 程序可进行查询与阻塞操作，或设置中断来捕获异步到达的信号。

通过写 SPE 的 MFC 中的 MMIO 寄存器，PPE 向 SPE 发送信号通知消息，该信号消息存放在 MMIO 寄存器中，SPE 程序通过读通道指令来读取信号通知数据。

(1) SPU 信号通道

每个 SPU 均有两个信号通知通道，分别对应于两个信号通知 MMIO 寄存器。一个信号的信息长度从 1 位到 32 位不等。一个 SPU 读信号通道时，如果没有信号可以读取，则 SPU 将会被阻塞。这两个信号通道中的每个信号通道均有一个数据项。这样，读通道计数器的返回值可表明是否有可用的信息存在。如果通道返回值为‘1’，则有信号存在。如果返回值为‘0’，则没有信号。如果信号通知通道的通道计数器的值为‘0’，则该通道的读指令将阻塞 SPU，直到有信号通知出现为止。

(2) 信号使用

像邮箱一样，当 SPE 将计算结果通过 DMA 方式存入主存时，信号通知通道非常有用。SPE 申请 DMA 传送之后，等待 DMA 传送操作完成，发送一个信号通知 PPE 它负责的计算任务已完成。在这种情况下，等待 DMA 操作完成时只能保证 SPE 的本地存储缓冲区可被重新使用，不能保证计算结果已保持一致性地写入主存。

2.4.3 Cell BE 设计特点

Cell BE 处理器所获得的优异性能在于其成功的架构设计，克服了处理器功耗、内存

访问和处理器频率对处理器性能提升的束缚。Cell BE 采用了异构多核架构(一个 PPE 负责运行操作系统和资源调度、8 个 SPE 运行计算密集型的应用程序),降低了电源功耗,提高了处理器的工作频率、并行性和安全性。Cell BE 采用的 3 层存储架构(即主存、SPE 本地存储器和 SPE 寄存器)解决了内存访问延迟问题。EIB 总线设计为各处理单元提供了快速的通信渠道。正是这些独特的架构设计,实现了 Cell BE 处理器在性能上的飞跃。

1. 异构结构设计

多核处理器体系结构有两个主要分支:同构多核和异构多核。同构多核处理器内部所有的核心结构完全相同,主要针对特征单一的应用,通过在多个处理器核上运行多个线程来挖掘更多的并行性,如 IBM 的 Power4;异构多核处理器内部芯片采用多种功能不同的核心,如负责管理调度的主核和负责计算的从核。Cell BE 处理器是一种典型的片上异构多核处理器,它由 1 个 PPE 和 8 个 SPE 通过片上宽带引擎组合而成。其中,PPE 运行操作系统,负责系统的全局控制。SPE 优化运行计算密集型任务。这种异构设计,能够使 PPE 和 SPE 工作在高频率状态而没有额外的负载,主频达到 3.2GHz,32 位峰值速度为 256GFLOPS,64 位峰值速度为 26GFLOPS。这种异构设计,优化配置了片上资源,不仅提升了处理器的执行效率,还降低了处理器的功耗。

2. 电源功耗设计

在 Cell BE 处理器设计之初,功耗和散热设计就是两个关键性问题。

提高电源功效的途径之一是采用异构架构:PPE 负责管理工作,SPE 负责运算工作。另外一个途径就是采用功耗管理单元(Power Management Unit,PMU)。当不需要芯片的全部运算能力时,可以用软件进行控制。PMU 允许操作系统对一个/多个单元甚至整个芯片进行调速、暂停、停止等操作,从而达到管理芯片功耗的目的。

在散热监控设计上,部署了热敏传感器和硬件控制的散热管理单元(Thermal Management Unit,TMU)。一个线性二极管把这两个模块的管脚连接起来,允许外部设备监控处理器的温度。该热敏传感器被放置在温度相对稳定的位置,读取处理器的整体温度,来控制外部的冷却装置。此外,还有 10 个数字热敏传感器(Digital Thermal Sensors,DTS)分布在芯片上,其中每个处理单元中都有一个 DTS,监测关键区域的温度。

TMU 不间断地监测每个热敏传感器,可通过编程来动态地控制每个处理单元的温度。当传感器达到指定的温度时就中断 PPE 的运行,处理单元的运行取决于与之相关联的 DTS 的温度。

通过软件设置 TMU 中每个传感器的 4 个温度值和调速量来控制 TMU。当温度上升时,第一个温度值指明处理单元什么时候结束调速,第二个温度值指明什么时候开始调

速，第三个温度值指明什么时候处理单元完全停止运行，第四个温度值指明什么时候芯片的时钟完全关闭。前两个温度值的设置有一定的滞后，以避免芯片频繁进行调速。当只靠调速不能完全控制芯片温度时，第三个温度值的设置将使芯片停止工作。只要温度值在第三个指定值之下，处理单元就能满足应用程序的实时性要求。当温度值超出第四个指定的安全操作温度值时，TMU 将自动关闭芯片时钟，避免芯片的永久性损坏。

3. 存储访问设计

Cell BE 处理器的存储结构是一种三层存储结构，分别是主存、SPE 本地存储器和 SPE 寄存器文件。PPE 通过 Load 和 Store 指令完成寄存器文件与主存之间的数据交换。通过 DMA 完成主存与 SPE 本地存储器之间的数据交换。SPE 通过访问本地存储来获取、载入和存储指令。

这种三层存储结构从根本上摆脱了传统的存储结构和编程模式，它在计算、数据交换和指令方面实现了并行，能够实现数据存取与计算之间的重叠，大大隐藏了存储访问延迟。由于大部分的数据交换都是在 SPE 的本地存储器与主存之间完成，因此 DMA 方式可大大降低因内存数据存储导致的程序数据延迟。

每个 SPE 支持 16 路 DMA 同时传输，因此，Cell BE 处理器可同时支持 128 路的 DMA 传输，差不多超过传统处理器带宽的 20 倍。同时，SPE 没有使用 cache 而使用本地存储器，虽然 cache 与本地存储器都是采用 SRAM 实现。但是，本地存储器是静态映射，其内部数据可直接通过特定的地址空间访问。而 cache 中的内容是动态映射，无法在指令中使用特定地址直接访问 cache。由于 SPE 没有 cache，它的性能将不受 cache 缺失的影响。这种存储结构能够保证高效的数据传输效率，其 I/O 与存储总带宽超过 100GB/s。

4. EIB 设计

EIB 是 Cell BE 处理器中通信结构的核心部分，用于完成 PPE、SPE、主存以及外部 I/O 之间的通信。EIB 为命令和数据设有分离的通信通道，每个总线单元通过点对点的方式连接到地址集中器，地址集中器从总线单元接收命令并将命令排序，按顺序把命令广播给所有的总线单元(用于监听)，然后聚集和广播命令响应，命令响应是某个单元开始数据传输的信号。

EIB 数据网络由 4 个 16B 宽的数据环组成：两个按顺时针运行、另外两个按逆时针运行。在通道不重叠的情况下，每个数据环最多能允许 3 个数据并发传送。初始化一个数据传送，总线单元必须请求数据总线访问。EIB 数据仲裁单元处理这些请求，决定哪个数据环应该处理哪一个请求。该数据仲裁单元总是选择两个数据环中传输距离最短的那个数据环，这样做可确保数据不用传送整个行程的一半就能到达目的地。数据仲裁单元

也可对传输进行调度以确保它不会涉及其他的传输。为了使读操作阻塞最小化，数据仲裁单元将较高的优先级赋予给来自内存控制单元的请求，对来自其他单元的请求则采取循环的方式。

EIB以处理器的一半时钟频率运行，在每个总线周期内，每个EIB单元能同时发送和接收16B的数据。EIB的最大数据带宽受系统内各个单元监听地址数量的限制，在每个总线周期内，只能监听一个地址。每个监听地址的请求最大能传送128B，Cell BE处理器的时钟主频为3.2GHz，所以理论上EIB的数据带宽峰值为128B·3.2GHz/2=204.8GB/s。

5. 安全性设计

CBEA安全架构的主要思想是：从系统中隔离出某个SPE并锁住其本地存储器(LS)，只有当前SPE可以使用，所有外部对本SPE的操作均无效。所有互连总线(EIB)上的PPE、其他SPE以及I/O发出的对该被隔离SPE的本地存储器的读写请求均不会对被锁住的本地存储器造成影响。一旦某个SPE被隔离，唯一的外部行为就是取消对它的隔离。此时，在外部访问再次有效之前，本地存储器和SPE中的所有数据都将被擦除。

所有这些操作均由硬件而非使用软件(比如，在一个表格中设置保护位等)实现。由于是完全的硬件隔离，所以，即便是操作系统或系统程序都不能访问被锁住的本地存储器和控制SPE的内核，黑客即使获得了根权限或系统权限也不能对在被隔离的SPE上执行的程序构成威胁。

2.4.4 发展情况与典型实例

2008年5月，IBM、Sony、Toshiba共同发布了第二代Cell BE芯片——PowerXCell 8i处理器。第一代Cell BE处理器的单精度浮点运算性能可达256GFLOPS，但一旦扩展为双精度，其运算性能即大幅下滑为25GFLOPS。PowerXCell 8i弥补了这一缺陷，使用新的扩展双精度SPE核心(eDP SPE)，在保持每个核心256KB缓存容量不变的情况下，PowerXCell 8i处理器的双精度浮点性能达到了第一代Cell BE处理器的5倍，更加适合超级计算机使用。

Cell BE处理器除了用于PS3之外，在刀片服务器上也得到了广泛的应用。Cell BE引领了刀片服务器的新时代，刀片服务器利用Cell BE处理器来加快计算密集型问题的处理速度，以满足一些特殊行业的需求。IBM分别于2006年和2007年推出了第一代和第二代的刀片系统QS20和QS21，并于2008年5月推出了第三代刀片系统QS22。QS22搭载了两个PowerXCell 8i处理器(3.2GHz主频)，最大可支持32GB内存，支持SAS硬盘，同时可支持8GB的固态硬盘、双千兆网络或Infiniband高速网络接口。

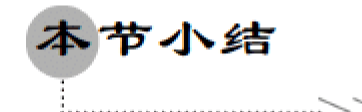
刀片服务器将Cell BE的应用领域扩展到医疗影像、航空航天、防务、数字动画、通信以及石油和天然气等行业，大大改变了这些行业的面貌。以下是一些具体的实例。

医疗行业：刀片服务器可大大减少医生比较和投射三维医疗影像所需的时间，这些影像通常历经数月甚至数年的积累，采用不同的分辨率，且来自多种不同的设备。在刀片服务器上运行医疗影像应用可帮助医生在数秒而非数分钟内投射多幅医疗影像，这不仅提高了医疗影像投射的精度，而且减少了诊断时间和病人的焦虑。

航空航天和防务领域：航空航天领域中的信号处理和雷达结果需要将位置、地形与速度、精度等结合在一起。运行在刀片服务器上的雷达应用和解决方案可提高雷达输出的保真度和分辨率，这意味着操作员可通过现有雷达系统看到之前从未看到过的物体和信息。

石油和天然气行业：石油公司可通过地震成像的最新技术，在更短的时间内以更高的精度对石油进行定位，大大提高了钻井的成功率，从而获得丰厚的利润。另一方面，降低了钻井的失误率，也意味着节省了钻井成本。

另外，Cell BE 在一些图像处理系统中也得到了应用。2008 年，丽台(Leadtek)推出了源于 Cell BE 芯片的视频编/解码加速卡。2010 年 7 月，Toshiba 推出了采用第二代 Cell BE 处理器的 3D 高清电视，搭载了 240Hz 的四倍速技术和全新的 3D 超解像技术，在观看普通清晰度 3D 节目源时提升了清晰度表现，其原理虽与之前的技术类似，但是在数据的处理能力上，却有了数倍的提升。



本节主要介绍了典型的异构多核处理器 Cell BE 的系统结构、cache 管理、片内通信以及其主要特点和发展情况。Cell BE 由 1 个 PPE 和 8 个 SPE 组成，PPE 负责运行操作系统、管理系统资源、完成任务分配和 SPE 调度，它包含一个 64 位的 PowerPC 处理器单元、两个独立的 32KB 大小的一级指令缓存和一级数据缓存以及一个共享的 512KB 的二级缓存。SPE 是基于 SIMD 精简指令集的加速核心，由双发射流水线 SPU 与 MFC 组成，拥有一块 256KB 的私有存储和一个 128×128 位的寄存器文件。PPE 与 SPE 之间通过 EBI 连接起来，并可通过 BEI 与 I/O 设备进行通信或对多 Cell BE 处理器进行互连。Cell BE 处理器独特的架构设计，突破了处理器发展的内存壁垒、功耗壁垒和频率壁垒，为处理器在异构多核方向的发展奠定了基础。

2.5 超级计算机

1993 年，在德国曼海姆大学，Hans Meuer 和 Erich Strohmaier 创建了世界上最权威的超级计算机排名榜——全球超级计算机 TOP500。TOP500 以超级计算机系统的 Linpack 测试值为基准进行排名，每年发布两次。超级计算机代表了一个国家在计算机

研发和应用方面的最高水平，所以每次 TOP500 的公布都可以显示一个国家在高性能计算方面的科研实力。

简单来说，超级计算机就是计算机中功能最强、运算速度最快、存储容量最大的一类计算机，通常是指由数千个甚至更多的处理器（机）组成的、能运算普通 PC 和服务器不能完成的大型复杂课题的计算机。在高运算速度前提下，人们可通过数值模拟来预测和解释以前无法进行实验的重大问题。许多重要的应用领域（如密码破译、武器研制、高精度气象预报、地球系统模式以及新材料研究等）都对使用高性能计算机提出了强烈需求。

世界上，许多发达国家的政府都大力支持研究机构开发高性能计算机。从 TOP500 排名的历史来看，美国、英国、法国、德国、日本都是高性能计算机研发、应用的传统强国。而近年来，印度、韩国也加大了对高性能计算机的支持力度，有多台计算机系统进入 TOP500 榜单。我国也加大了对高性能计算机的支持力度，特别是在第 36 届 TOP500 排名中，天河一号二期系统一举超过美国橡树岭国家实验室的"美洲虎"超级计算机而雄踞首位。同时在第 36 届 TOP500 排名中，中国入选了 41 台高性能计算机，仅次于美国。这都体现了我国在高性能计算机领域较强的研发能力。

2.5.1　超级计算机的发展与规律

在 20 世纪 70 年代出现的向量计算机可被看作是第一代的高性能计算机。通过在计算机中加入向量流水部件，可大大提高科学计算中向量运算的速度，其中比较著名的有 CDC 系列、CRAY 系列、NEC 的 SX 系列向量机。我国有代表性的是由中国科学院计算技术研究所（简称为中科院计算所）研制的 757 计算机（如图 2-42（a）所示）和由中国国防科学技术大学研制的银河一号计算机（如图 2-42（b）所示）。



(a) 中科院计算所研制的757计算机



(b) 中国国防科学技术大学研制的银河一号

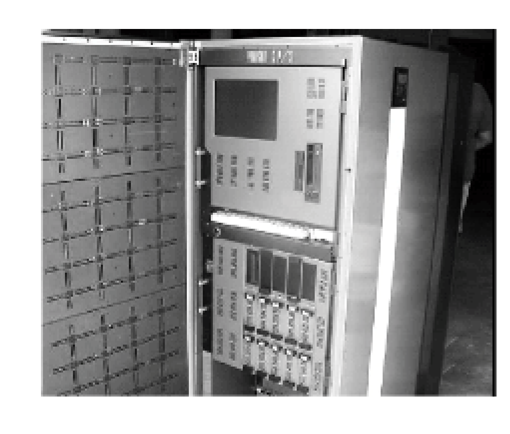
图 2-42　中国第一代向量计算机

20 世纪 80 年代初，随着 VLSI 技术和微处理器技术的发展，向量机一统天下的格局逐渐被打破。从成本上来说，由多个廉价微处理器组成的并行化超级计算机具有无可比拟的优势。"性能/价格比"而非单一性能成为衡量高性能计算机系统的重要指标。按照

摩尔定律，微处理器的性能快速超越了传统的向量机。

20 世纪 90 年代初，大规模并行处理（MPP）系统已开始成为高性能计算机发展的主流。MPP 主要由多个处理器通过高速互联网构成，处理器之间通过消息传递的方式进行通信与协调。具有代表性的是 TMC CM-5（如图 2-43（a）所示）和 Intel Paragon（如图 2-43（b）所示）。我国第一台 MPP 系统是由中科院计算所国家智能机中心研制的曙光 1000（如图 2-43（c）所示）。比 MPP 系统早几年问世的对称多处理机 SMP 系统，是由数目相对较少的处理器共享物理内存和 I/O 总线形成的计算机系统，我国最早的 SMP 是由中科院计算所国家智能机中心研制的曙光 1 号。与 MPP 系统相比，早期的 SMP 系统的扩展能力有限，并不具有很强的计算能力。但由于 SMP 系统与单机系统的兼容性较好，是单机系统的升级与增强，在当时被广泛应用于商业计算领域。第一届 TOP500 排名中位列第一的超级计算机是 TMC CM-5，其由 1024 个处理器组成，具有每秒 600 亿次浮点运算能力。


(a) TMC CM-5


(b) Intel Paragon


(c) 曙光1000

图 2-43　MPP 系统

20 世纪 90 年代中后期，一种趋势是将 SMP 系统的优点与 MPP 系统的扩展能力相结合，从而发展成为后来的 CC-NUMA 结构，即分布式共享内存。其每个处理器结点均可访问到所有其他结点的内存，但访问远程内存时延迟相对较大。代表性的系统有 Sequent NUMA-Q、SGI-Cray Origin、我国的神威与银河系列等。在提高性能方面，CC-NUMA 结构本身并未进行较大的创新，其主要优点在于程序的开发和与 SMP 系统的兼容性。

在发展 CC-NUMA 结构的同时，机群系统（Cluster）也迅速发展了起来。类似 MPP 系统，机群系统是由高速网络将多个处理器互连而成，但其结点一般是可单独运行的商品化计算机。由于规模经济成本低的原因，机群系统具有比 MPP 系统更高的性能/价格比优势。同时，机群系统还继承了 MPP 系统的编程模型，更进一步加强了其竞争优势。具有代表性的机群系统有 IBM SP2、曙光 3000 和 4000（如图 2-44 所示）等。从 2000 年开始，机群系统实际上已构成了高性能计算机系统的主流，到了 2010 年 11 月，TOP500 排名中机群系统达到了 415 台。



(a) 曙光3000机群系统



(b) 曙光4000机群系统

图 2-44　曙光 3000 和曙光 4000 机群系统

2008 年，IBM 推出了以 Roadrunner(如图 2-45 所示)为代表的新型低功耗的高能效系统，运算速度首次突破了 1Petaflops(即千万亿次/秒)。Roadrunner 首次将传统的超级计算机处理器与专为 Sony PS3 所设计的 Cell 芯片相结合，这就意味着 IBM 首次将异构计算引入其超级计算机中。近年来，在高性能计算(HPC)领域中，与多核一起崛起的一个新的趋势就是异构计算，这在国际高性能计算领域又掀起了一阵热潮，被公认为提高 HPC 性能的有效手段。在 2008 年 6 月的 TOP500 排名中，也首次提供了计算机的节能排名。该节能排名主要是看计算机系统在运行一个典型 HPC 应用时的能耗情况，并没有考虑外部制冷、磁盘以及其他外部环境带来的能耗影响。在今后的计算技术发展趋势中，低能耗应该成为一个重要的衡量标准。在高性能计算机系统结构研究方面，许多国家都在积极进行新的尝试。



图 2-45　IBM Roadrunner 超级计算机

综上所述，从 20 世纪 70 年代至今，高性能计算机的体系结构主要经历了如下四次变革：

- 以向量机、SIMD 为代表的数据并行系统(Data Parallel System);
- 以 MPP、Scalar SMPs、Constellations 为代表的定制可扩展系统(Custom Scalar System);
- 以机群、刀片为代表的商业化机群系统(Commodity Cluster System);
- 以 BlueGene 系列为代表的高能效系统(Power Efficient System)。

从表 2-5 可以看出,高性能计算机的体系结构每隔十年就有一次重大的突破,而性能则相应地提高了一千倍。自 20 世纪 60 年代 CDC6600 推出以来,至今高性能计算机的性能已提高了约十亿倍。根据 TOP500 的历史数据进行预测,2019 年人类将进入艾级(ExaFlops,百万万亿次)计算时代。

表 2-5 高性能计算机体系结构的发展

阶段	萌芽阶段 (1960—1975)	向量机阶段 (1976—1989)	MPP 系统阶段 (1990—1999)	MPP+机群系统阶段 (2000—2011)
典型系统	CDC 6600 STAR-100 ILLIAC-IV	Cray-1/2/3 NEC sx-1/2/3	Intel Paragon TMC CM-5 Cray T3D Intel Option Red Blue Mountain	IBM BlueGene P IBM RoadRunner 国防科大的天河一号
性能	M FLOPS 量级	G FLOPS 量级	T FLOPS 量级	P FLOPS 量级

表 2-6 展示了一些高性能计算体系结构从出现到流行再到逐渐消亡的发展过程。从表 2-6 中可以发现一个规律,即每一种高性能计算机体系结构从早期出现到成为主流结构一般经历了 10 年,随后再经过 10 年逐渐被淘汰。以 MPP 系统为例,1976 年 ILLIAC IV 第一次将多个处理器进行互连实现并行处理;20 世纪 80 年代开始逐渐涌现出一些 MPP 高性能计算系统,包括 GoodyearMPP、CM 等;进入 20 世纪 90 年代 MPP 系统替代向量机成为高性能计算的主流结构,20 世纪 90 年代中期 MPP 系统在 TOP500 中的比例已超过 80%;随着机群的兴起,MPP 系统的比例又开始逐渐下降,到 2009 年 11 月,MPP 系统在 TOP500 中的比例只占到了 16%。

表 2-6 高性能计算机体系结构的演变

年度	1980	1990	2000	2010
体系结构	SIMD	SMP+S2MP	MPP	Cluster
	Single Processor	MPP	Constellations	MPP
	SMP	Single Processor	Cluster	Constellations

高性能计算机体系结构的变迁周期从某种程度上预示着一个新趋势：性能/功耗比更高的高能效计算机，将从 2010 年开始替代自 20 世纪 90 年代兴起的机群系统，而成为高性能计算体系结构的主流，这种趋势将持续到 21 世纪 20 年代中期。因此，以 BlueGene、RoadRunner 等为代表的高能效计算机在 2020 年之前仍将是高性能计算机的主流体系结构，预计在 2019 年左右出现的艾级超级计算机很可能还会采用这种结构。

2.5.2 超级计算机的现状

2010 年 11 月 16 日，在美国新奥尔良举行的 SC10 大会上，发布了第 36 届 TOP500 排名。经过技术升级的我国"天河一号"二期系统（天河-1A，如图 2-46 所示）一举超过美国橡树岭国家实验室的"美洲豹"超级计算机而雄踞首位。"天河一号"系统于 2009 年 10 月底由中国国防科学技术大学研制，于 2010 年在中国国家超级计算天津中心安装部署，其问世标志着中国成为继美国之后，第二个能够研制千万亿次超级计算机的国家。



图 2-46 天河一号超级计算机

在第 36 届 TOP500 排名前十的计算机系统中，有七台计算机系统的运算能力超过了千万亿次，其中五台为新上榜的机器。在排名前十的计算机系统中，有五台计算机系统是由美国公司制造，其他分别来自中国（两台）、日本（一台）、法国（一台）以及德国（一台）。图 2-47 列出了第 36 届 TOP500 排名前十的超级计算机。

在本届 TOP500 排名的计算机系统中，中国从上一届入选的 24 台跃升至本届的 41 台，排名一举超过日本、法国、德国和英国而跃居第二，仅次于美国。美国依然处于 HPC 系统的领导地位，在本届 TOP500 排名中占有 275 台，低于上一届的 282 台。欧洲共占有 124 台，低于上一届的 144 台，但仍多于亚洲。在欧洲，法国和德国赶上了英国，英国从上一届的 38 台下降至本届的 25 台，法国本届占有 26 台（上届 29 台），德国占有 26 台（上届 24 台）。亚洲共占有 84 台，高于上一届的 57 台。其中，中国占有 41 台（在亚洲占主导地位，上届 24 台），日本占有 26 台（上届 18 台），印度占有 4 台（上届 5 台）。

Rank	Site	System	Cores	R_{max}	R_{peak}
1	National Supercomputing Center in Tianjin China	NUDT TH MPP, X5670 2.93Ghz 6C, NVIDIA GPU, FT-1000 8C NUDT	186368	2566	4701
2	DOE/SC/Oak Ridge National Laboratory United States	Cray XT5-HE Opteron 6-core 2.6 GHz Cray Inc.	224162	1759	2331
3	National Supercomputing Centre in Shenzhen (NSCS) China	Dawning TC3600 Blade, Intel X5650, NVidia Tesla C2050 GPU Dawning	120640	1271	2984.3
4	GSIC Center, Tokyo Institute of Technology Japan	HP ProLiant SL390s G7 Xeon 6C X5670, Nvidia GPU, Linux/Windows NEC/HP	73278	1192	2287.63
5	DOE/SC/LBNL/NERSC United States	Cray XE6 12-core 2.1 GHz Cray Inc.	153408	1054	1288.63
6	Commissariat a l'Energie Atomique (CEA) France	Bull bullx super-node S6010/S6030 Bull SA	138368	1050	1254.55
7	DOE/NNSA/LANL United States	BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz, Voltaire Infiniband IBM	122400	1042	1375.78
8	National Institute for Computational Sciences/University of Tennessee United States	Cray XT5-HE Opteron 6-core 2.6 GHz Cray Inc.	98928	831.7	1028.85
9	Forschungszentrum Juelich (FZJ) Germany	Blue Gene/P Solution IBM	294912	825.5	1002.7
10	DOE/NNSA/LANL/SNL United States	Cray XE6 8-core 2.4 GHz Cray Inc.	107152	816.6	1028.66

图 2-47　第 36 届 TOP500 排名前十的超级计算机

新上榜的超级计算机与图形处理器的发展不无关系，这再次说明了异构计算是未来超级计算机的重要发展方向。表现出众的天河一号(中国)和 TSUBAME 燕 2.0(日本)都使用了 NVIDIA 图形处理器来加快运算速度。在本届 TOP500 排名的计算机系统中，有 17 款计算机系统使用了图形处理器。其中，六款使用的是 Cell 处理器，10 款使用的是 NVIDIA 处理器，还有一款使用的是 ATI Radeon 处理器。

在超级计算机体系结构方面，本届 TOP10 中 Cluster 结构和 MPP 结构各占有 5 台机器，平分秋色。Cray 的“美洲豹”、“跳跃者”、“海怪”和“天空”以及 IBM 的“Blue Gene/PSolution”都采用了 MPP 结构；中国国防科学技术大学的“天河一号”、曙光的“星云”、日本的“TSUBAME 燕 2.0”、法国的“BullTera-100”和 IBM“走鹃”系统都采用了 Cluster 结构。在本届 TOP500 超级计算机系统中，Cluster 结构仍是主要使用的结构，但所占数量略有下降，本届占有 415 台(上届为 424 台)。采用 MPP 结构的系统数量略有增加，从上届的 74 台增加到 83 台，虽然 MPP 结构的系统在 TOP500 中占有的数量不多，但是它主打高端市场。

在本届 TOP10 排行榜中，Cray“跳跃者”采用十二核处理器，IBM“走鹃”采用九核处理器，Cray“天空”与“Bull Tera-100”采用八核处理器，Cray“美洲豹”与“海怪”、“天河一号”、“星云”和“TSUBAME 燕 2.0”都采用六核处理器，余下的一台采用了四核处理器。

由此可见,在超级计算机系统中,四核处理器将逐渐被淘汰,六核处理器趋于主流,而六核以上处理器趋于流行。多核技术能够使服务器并行处理任务,多核系统更易于扩展,并且能够在更纤巧的外形中融入更强大的处理性能,其所需的功耗更低、产生的热量更少。

在本届 TOP10 的排名中,来自中国的两台超级计算机系统“天河-1A”与“星云”和由日本东京工业大学研制的“TSUBAME 燕 2.0”系统都采用了 NVIDIA GPU 来提升系统的整体性能。在本届 TOP500 中取得第一位的天河-1A 超级计算机就采用了多核心 CPU 与 GPU 相结合的异构架构,在性能、功耗、占地面积上取得巨大的突破。从“天河-1A”与“星云”在 TOP500 所处的领先位置以及日本“TSUBAME 燕 2.0”系统的加入,可以看出在高性能计算领域中 CPU+GPU 的混合架构越来越流行。

在本届 TOP500 中,功耗已成为评价一个系统性能和架构趋势的关键性指标。随着功耗的增加,系统的计算效率也在提高。对于现代超级计算机来说,节能和性能同等重要,而且二者完全可以共存。计算密度更高、功耗更低、性能更强的超级计算机正在不断地涌现。在此之前,超级计算机几乎成为高功耗的代名词,每年的运营成本极高,随着厂商对低碳的重视,近年来出现的系统几乎都基于低功耗设计。通过这些设计,不仅可以降低能耗,提高每瓦特性能,而且减少了在制冷方面的支出。在本届 TOP500 中,系统功耗方面的统计信息如下:

- 25 台计算机系统的功耗大致在 1MW;
- IBM 新的 BlueGene/Q 计算机系统原型创造了 1680MFLOPS/W 功耗效率的新记录;
- 在 TOP500 的超级计算机系统中,平均功耗为 477KW,平均功耗效率为 195MFLOPS/W,相比一年之前的 150MFLOPS/W 有所上升;
- 在 TOP10 的超级计算机系统中,相比 6 个月前,平均功耗由 2.89MW 缓慢提升为 3.2MW,平均功耗效率由 300MFLOPS/W 略微下降为 268MFLOPS/W。

2.5.3　超级计算机面临的挑战

按照过去的趋势来推算,在 2019 年应能出现峰值速度超过每秒百亿亿次运算(EFLOPS,1018FLOPS)的系统。但如果按传统方式采用通用 CPU 等部件来构建百亿亿次高性能计算机,将遇到功耗、成本、可用性等多方面的重大挑战。

LLBL 进行的一项研究表明,使用现有的 Cluster 技术构建 200PFLOPS 的系统,如果使用 AMD Opteron CPU(处理器频率为 2.8GHz)的话,将需要 18 亿美元的成本,功耗为 175MW。如果使用 IBM 的 BlueGene/L(处理器频率为 700MHz)的话,成本也高达 26 亿美元,功耗为 27MW。

高达数十亿美元的成本将使得 EFLOPS 计算机的构建面临巨大的挑战。与此同时,数十至数百兆瓦的功耗也成为高性能计算机在部署和使用过程中的重大障碍。由于耗电

量巨大,甚至需要为高性能计算机单独设置发电站和供电线路。巨大的系统功耗还对散热系统提出了极高的要求。所有这些因素,都将进一步增加高性能计算机的部署成本和使用成本。

综上所述,要研制性能达到 EFLOPS 的下一代高性能计算机,就必须对现有的计算机系统结构进行重大变革。

1. 众核技术

利用众核技术构造低功耗处理器是目前国内外开展低功耗体系结构研究的主流方式。该技术的基础是:芯片在高频区间的动态功耗与频率的三次方近似成正比。因此,通过众核技术(虽然单个核的主频较低,但是多个核并行执行)可得到较好的系统性能,同时较低的主频使得众核芯片的功耗较单核高主频芯片要低很多。根据众核芯片在整个计算系统中的使用方式,可将众核技术分为同构众核和异构众核两种方式。

1) 同构众核技术

IBM 的 BlueGene 系列芯片是低功耗同构众核处理器的典型代表。BlueGene/L 已被 IBM 用于构建多台 PFLOPS 级的系统。与此同时,IBM 还启动了 BlueGene/C 计划(又称 Cyclops64)。Cyclops64 处理器在一个芯片内封装了 80 个处理器核心,工作频率为 500MHz,每个核心包括一个 64 位的浮点运算单元和两个线程单元,使得每个核心可同时执行两个线程,整个芯片的峰值速度可达 80GFLOPS。一个完整的 Cyclops64 系统由 24×24×24 个芯片连接而成,峰值速度为 1.1PFLOPS。可以看出,与 BlueGene/L 相比,Cyclops64 在单个芯片内集成了更多的芯片,大大减少了构建高性能计算系统所需的芯片数,从而可有效降低成本和功耗。Cyclops64 的体系结构如图 2-48 所示。

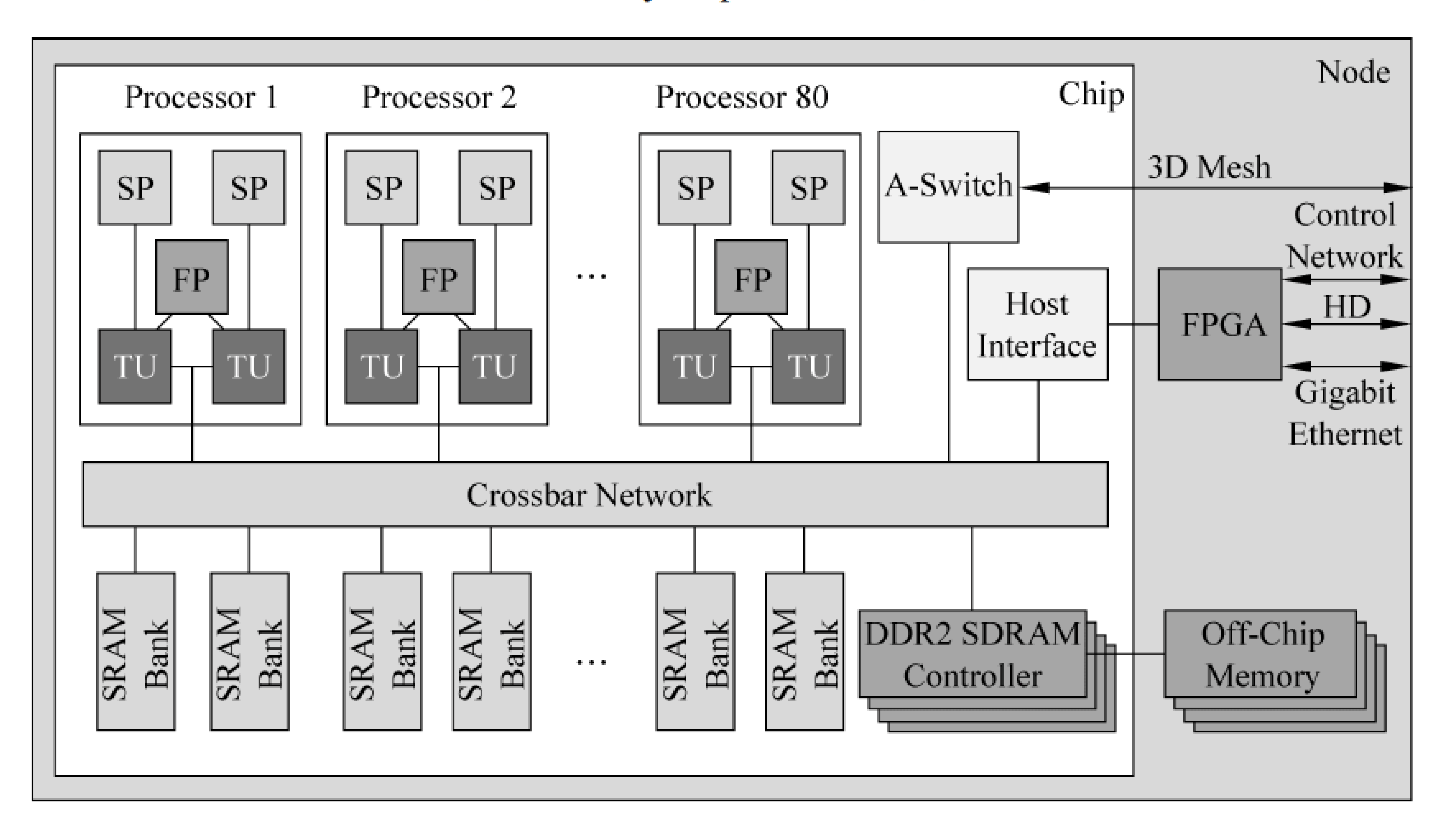


图 2-48 Cyclops64 的体系结构

与目前主流的通用处理器(例如,Intel 5650 处理器,2.66GHz,六核,峰值速度为64GFLOPS,功耗为 95W)相比,众核处理器的频率较低,但核数较多,它利用向量处理单元提高峰值速度,处理芯片的性能/功耗比得到了显著提高。

同构众核技术的优势在于对现有应用具有较好的继承性,单个核上基本都可以运行操作系统(或简化的操作系统),并支持 MPI 编程模型,从而使现有的大量 MPI 应用程序不用修改或稍加修改即可在这类系统上运行。

2) 异构众核技术

目前主流的异构多核/众核处理器主要有如下两类:

- 一类是 IBM 的 Cell BE 芯片,采用 1 个通用的 PPE 和 8 个向量 SPE,用于高性能计算的 PowerXCell 8i 可达到 102 GFLOPS 的双精度浮点性能,世界上首台超 PFLOPS 的系统就是 IBM 基于 Cell BE 和 AMD Opteron 平台混合构建的 RoadRunner 系统;
- 另一类异构众核处理器是图形处理器(GPU)。自从 NVIDIA 推出 CUDA 语言以来,使用图形处理器进行通用编程得到了广泛接受。最新的 GPU,如 NVIDIA Fermi 已克服了第一代 GPU 的部分缺陷(如双精度浮点计算能力较差,内存访问没有 ECC 校验等),以其相对低廉的成本成为构建高性能计算机的一种流行方式。曙光的星云计算机就是通过使用 NVIDIA Fermi 图形处理器,获得了超 1PFLOPS 的实测 Linpack 性能,成为目前(第 36 届)Top500 上排名第三的系统。

除了以上两种主流的异构多核加速器芯片外,还有诸如 ClearSpeed、Tilera64 等加速器,但由于产量较低、价格相对较贵,并未得到广泛的应用。

异构众核技术,特别是 GPU 技术,由于其产量巨大带来的规模成本优势将成为未来一段时期构建高性能计算机的流行方式。但是这种方式也有其自身的缺点:

- 应用开发困难。必须要对原有的高性能计算应用进行显式划分,确定哪些部分在主 CPU 上运行,哪些部分在 GPU 上运行。
- 划分到 GPU 上的任务,其编程比较困难。由于 GPU 上的“核”通常比较简单(例如,NVIDIA GPU 上的“核”更类似于一个向量处理单元),而且 GPU 还不支持动态内存分配和锁操作等。因此,对于 GPU 上的程序设计与优化比起通用 CPU 来说更困难。

因此,尽管 GPU 加速器技术可以提供很好的 Linpack 性能,但如何提高 GPU 的可编程性,将是关系到使用 GPU 来构建未来高性能计算机系统的关键问题之一。

2. 新的半导体工艺

目前,希望在器件方面取得突破,以期做出更快、更省电、集成度更高的芯片。在处理器工艺上,国际上对可能替代 CMOS 的其他集成电路技术(例如,光学芯片、分子芯片等)

进行了探索，但是现有的新工艺技术与CMOS相比，并不具有显著的优势。

国际上，学术界和工业界对内存新工艺的研究也非常活跃，例如，铁存储器(FeRAM)、磁存储器(MRAM)、相变存储器(PCRAM)以及忆阻器(Memrisistor)等方面的研究。从目前的发展来看，磁存储器和相变存储器的发展更迅速，三星公司已推出了相变内存产品芯片。磁存储器和相变存储器的特点是集成度高和非易失性，即不需要通过刷新电流来维持系统状态，其漏电功耗很低，是构成下一代高性能计算机存储系统的很有前途的器件。不过，目前这些新内存在可写寿命、写入功耗、写入速度等方面还与目前广泛使用的SRAM有着比较明显的差距。

另一项对高性能计算机有重大影响的半导体工艺技术是集成电路的三维封装技术。该技术可在一个芯片内封装更多的处理器核，或直接将部分内存封装在处理器内以缓解多核、众核处理器带来的内存带宽瓶颈问题。美国宾夕法尼亚州州立大学与HP实验室合作，将PCRAM与DRAM封装起来，可进行快速检查点(checkpoint)设置，同时还提出了一种利用先进半导体工艺来解决高性能计算机面临的可用性挑战问题的方法。

3. 定制化的高性能处理器

根据应用的不同需求，定制面向一类问题的多用或专用处理器，能够设计出具有更高性能/功耗比的处理器。但是，高性能计算机的构建并不只是研制出处理器芯片，还需要针对处理器开发相应的系统软件和工具链(包括编译器、链接器、调试器、操作系统等)。采用传统技术，完成定制芯片和相应的系统软件和工具链，需要较长时间和很高的成本，往往还不如半导体技术发展带来的性能提高更有效，因此如何做到处理器的快速定制是这条研制路线的关键。

目前，Tensilica公司在处理器定制技术方面具有较好的基础。其采用基于Open64的高性能编译器，可让用户方便地定制所需的全套软件工具和芯片，为执行特定类型的应用提供远高于通用CPU的性能。目前，已实现一个面向网络处理的188个核心的处理器，并用于Cisco的路由器中。Tensilica公司开发出了新一代处理器定制技术，可全自动地完成芯片的硬件描述语言、系统软件以及全套工具链的生成。美国LBNL已决定采用Tensilica的定制技术来构建其EFLOPS级高性能计算机。

本节小结

本节主要介绍TOP500超级计算机系统排名。超级计算机代表了一个国家在计算机研发和应用方面的最高水平，显示一个国家在高性能计算方面的科研实力。我国的“天河-1A”夺得了第36届TOP500排名的第一名，与图形处理器的性能提升不无关系，这也

说明异构计算是未来超级计算机的重要发展方向。同时，Cluster 结构的系统在 TOP500 中比例较高，而 MPP 结构的系统在 TOP500 中占有数量虽然不多，但其主打的是高端市场。

参考文献

1　张林波，迟学斌等. 并行计算导论. 北京：清华大学出版社，2006.
2　陈国良，吴俊敏等. 并行计算机体系结构. 北京：高等教育出版社，2002.
3　张晨曦，王志英等. 计算机系统结构教程. 北京：清华大学出版社，2009.
4　徐炜民，严允中. 计算机系统结构. 北京：电子工业出版社，2010.
5　多核系列教材编写组. 多核程序设计. 北京：清华大学出版社，2007.
6　张林波，迟学斌等. 并行计算导论. 北京：清华大学出版社，2006.
7　李宝峰等译. 多核程序设计技术：通过软件多线程提高性能. 北京：机械工业出版社，2007.
8　武汉大学多核架构与编程技术课程组. 多核架构与编程技术. 武汉：武汉大学出版社，2010.
9　濮元恺. 多核 CPU 展望. http://www. st-tech. net/csck.
10　NVIDIA 官方网站. http://www. nvidia. com.
11　张舒，褚艳丽等. GPU 高性能运算之 CUDA. 北京：中国水利水电出版社. 2009.
12　Rob Farber. NVIDIA Announces New Tesla Parallel Processor Cards and GTX200 Series Graphics Cards. http://drdobbs. com/high-performance-computing/208404203.
13　John D. Owens，David Luebke，et al. Survey of General-Purpose Computation on Graphics Hardware. Computer Graphics Forum，2007，26(1)，80～113.
14　KIBM Inc. Cell Broadband Engine Programming Handbook v1. 0. New York：IBM Press，2006：25～210.
15　Kahle J. A.，et al. Introduction to the cell multiprocessor. IBM Journal of Research and Development，2005，49(4. 5)：589～604.
16　IBM Inc. Cell Broadband Engine Architecture. New York：IBM Press，2006：20～123.
17　Gschwind M，et al. An Open Source Environment for Cell Broadband Engine System Software. Computer，2007，40(6)：37～47.
18　林海波，谢海波等. Cell BE 处理器编程指南. 北京：电子工业出版社，2008：15～242.
19　黄国睿，张平等. 多核处理器的关键技术及其发展趋势. 计算机工程与设计，2009，30(10)：2414～2418.
20　Kumar R，Farkas K I，et al. Single-ISA heterogeneous multi-core architectures：The potential for processor power reduction. Proceedings of the 36th International Symposium on Microarchitecture. Los Alamitos：IEEE Computer Society，2003. 81～92.
21　Pham，D. C.，et al. Overview of the architecture，circuit design，and physical implementation of a first-generation cell processor. IEEE Journal of Solid-State Circuits. 2006，41(1)：179～196.
22　Kistler，M.，M. Perrone，et al. Cell Multiprocessor Communication Network：Built for Speed. IEEE Micro，2006，26(3)：10～23.

23 Shimizu K, Brokenshire D, et al. Cell Broadband Engine Support for Privacy, Security, and Digital Rights Management Applications. GSPx 2005,23(2): 120～123.
24 Cell Broadband Engine resource center. https://www.ibm.com/developerworks/power/cell/.
25 Top500 官方网站. www.top500.org.
26 戴坚君,杜晓梅. 第 36 届 TOP500 超级计算机浅析. 高性能计算发展与应用,2010,33(3),72～77.
27 谢向辉,方兴. 超级计算机的演进趋势. 高性能计算发展与应用,2010,33(3),17～25.

CHAPTER 3 第3章　并行编程模型与语言

3.1　MPI

并行编程模型发展迅速，现存有各种并行编程语言，本节将对目前流行的 MPI 编程模型进行详细的介绍。

基于消息传递的 MPI，是目前应用最为广泛的并行编程模型之一。本节将详细讲解 MPI 并行编程的特点，分析常用的 MPI 函数和编写 MPI 并行程序的方法。通过本节的学习，读者对并行编程模型会有一定的了解，并能掌握基本的并行编程方法。

3.1.1　MPI 简介

1. 什么是 MPI

MPI 是一种标准或规范的代表，而不是特指某一个对它的具体实现。所有正确的 MPI 程序，都可以不加修改地运行在任意一台并行机上。MPI 是一种消息传递编程模型，并成为这种编程模型的代表和事实上的标准，它服务于进程通信。

MPI 是一种库描述，而不是一种语言。很多人认为 MPI 是一种专门的并行编程语言，这是不正确的。MPI 由 FORTRAN＋MPI 或 C＋MPI 组成，共有上百个函数调用接口，在 Fortran 和 C 语言中可以直接对这些函数进行调用。

由于 MPI 并行代码的易移植性，应用程序员不必掌握更多的全新概念，即可很轻松地编写 MPI 程序。

2. MPI 的发展历程

MPI 的标准化工作涉及大约 60 个国家的人们，他们主要来自于美国和欧洲的 40 多个组织，这包括大部分并行计算机的主要生产商，还有来自大学、政府实验室和工厂的研究者们。MPI 的标准化工作始于有关分布存储环境中消息传递标准的讨论会，该会议是由并行计算研究中心资助，于 1992 年 4 月 29 日至 30 日在威廉姆斯堡召开。会议上讨论了标准消息传递的必要的、基本的特点，并建立了工作组继续进行标准化工作。

由 Dongarra、Hempel、Hey 和 Walker 建议的初始草案 MPI-1 于 1992 年 11 月推出，并在 1993 年 2 月完成了修订版。在威廉姆斯堡讨论会上认定的消息传递标准的主要特

点都包含在 MPI-1 中。由于 MPI-1 的基本目的就是促进讨论并继续此项工作，所以其内容主要集中在点对点的通信。

1992 年 11 月，MPI 工作组在明尼阿波利斯召开会议，对 MPI 标准的各主要组成部分建立相应的分组委员会，并设定了目标：到 1993 年秋产生 MPI 草案。为达到该目标，MPI 工作组在 1993 年的前 9 个月中每隔 6 个星期就讨论两天，终于在 1993 年 11 月召开的超级计算会议上提出了 MPI 标准的草案。

1994 年 5 月，MPI 标准正式发布。

1997 年，MPI-2 正式发布。

3. 为什么要用 MPI

为什么要选择 MPI 呢？是因为 MPI 的高可移植性和易于使用。在以低级消息传递程序为基础的较高级和（或）抽象程序所构成的分布存储通信环境中，MPI 标准化所带来的效益非常明显。而且，消息传递标准的定义能提供给生产商清晰定义的函数库，以便生产商能有效地实现这些函数库或在某些情况下为库函数提供硬件支持，因此增强了 MPI 的可扩展性。

迄今为止，MPI 已在 IBM PC 上、MS Windows 上、所有主要的 UNIX/Linux 工作站、主流的并行机上得到实现。使用 MPI 进行消息传递的 C 或 Fortran 并行程序不用改变，就运行在 IBM PC、MS Windows、UNIX/Linux 工作站以及各种并行机上。

3.1.2 第一个 MPI 程序

1. MPI 环境的配置

1）单机上 MPI 环境配置

目前，市场上主流的个人计算机（PC）都是双核以上的硬件配置，下面将以双核 PC 为例进行 MPI 环境的配置。MPI 在并行执行的时候是通过 ssh 来实现结点间的通信，所以安装 MPI 时还需对 ssh 服务进行必要的配置。以下是安装步骤。

（1）配置 ssh

① 在当前用户下，执行 $ ssh-keygen -d，将提示密钥存放的目录，敲回车确定。提示输入密码，为了运行程序时方便，这里不需要输入密码，直接输入两次回车确认即可，如图 3-1所示。

```
[test@localhost ~]$ ssh-keygen -d
Generating public/private dsa key pair.
Enter file in which to save the key (/home/test/.ssh/id_dsa):
Created directory '/home/test/.ssh'.
Enter passphrase (empty for no passphrase): █
```

图 3-1 在当前用户配置 ssh

② $ cd ～/. ssh。

```
$ cp id_dsa.pub authorized_keys
```

将用户的共有密钥放在自己的认证密钥里，这样在 $ ssh localhost 时就不用输入密码了。但如果是在集群环境的话，则要将自己的共有密钥存放在要访问的其他结点的authorized_keys 中，如图 3-2 所示。

```
[test@localhost ~]$ cd ~/.ssh
[test@localhost .ssh]$ cp id_dsa.pub authorized_keys
```

图 3-2　认证共有密匙

③ 修改/etc/hosts 文件，从 localhost localhost. localdomain 下面一行开始，填入需要运行 MPI 的所有结点名及其 ip 地址：

```
(本机 ip)    node1
(本机 ip)    node2
(本机 ip)    node3
(本机 ip)    node4
```

如图 3-3 所示。

```
127.0.0.1      localhost localhost.localdomain localhost4 localhost4.localdomain4
192.168.126.136 node1
192.168.126.136 node2
192.168.126.136 node3
192.168.126.136 node4
::1            localhost localhost.localdomain localhost6 localhost6.localdomain6
```

图 3-3　修改后的/etc/host 文件

④ 修改(或创建)/etc/hosts. equiv 文件，将允许访问本结点进行 mpi 计算的所有其他结点填入，一行一个结点名。这一步是使本结点对其他结点放权，文件如下：

```
node1
node2
node3
node4
```

如图 3-4 所示。

```
node1
node2
node3
node4
```

图 3-4　/etc/host. equiv 文件

⑤ 测试一下。

```
$ ssh node1
$ ssh node2
```

如果不用输入密码的话，就说明 ssh 配置成功了。

(2) 安装 mpich

下载 mpich-1.2.7 下载地址为 http://www.mcs.anl.gov/research/projects/mpich2/（本书中采用的是 mpich-1.2.7 版本）

```
$ cd mpich-1.2.7pl
./configure --prefix=/usr/local/mpi -rsh=ssh --disable-weak-symbols
```

（注：使用选项--disable-weak-symbols 的目的是为了避免 MPICH 1.2.5 的一个 bug，否则在生成的库中将没有 MPI_File_xxxx 等函数，只有 PMPI_File_xxx 等函数）如图 3-5 所示。

```
[test@localhost Downloads]$ cd mpich-1.2.7p1/
[test@localhost mpich-1.2.7p1]$ ./configure --prefix=/usr/local/mpi -rsh=ssh --d
isable-weak-symbols
```

图 3-5 configure 配置相关文件

```
make
make install(注：这步需要 root 权限)
```

至此 mpich 安装完毕。

(3) 配置环境变量

分别将/usr/local/mpi/bin 和/usr/local/mpi/man 加入环境变量 PATH 和 MANPATH 中，只需在目录/etc/profile.d 中创建 mpich.sh 和 mpich.csh 这两个文件即可，它们分别对 Bourne shell 和 C shell 起作用，这两个文件的内容如下：

① mpich.sh 文件。

```
#!/bin/bash
export MANPATH=${MANPATH}:/usr/local/mpi/man
export PATH=${PATH}:/usr/local/mpi/bin
```

如图 3-6 所示。

```
#!/bin/bash
export MANPATH=${MANPATH}:/usr/local/mpi/man
export PATH=${PATH}:/usr/local/mpi/bin
```

图 3-6 mpi.sh 文件

② mpich. csh 文件。

```
#!/bin/csh
if ( $?MANPATH == 0 ) then
setenv MANPATH :/usr/local/mpi/man
else
setenv MANPATH ${MANPATH}:/usr/local/mpi/man
endif
setenv PATH ${PATH}:/usr/local/mpi/bin
```

如图 3-7 所示。

```
#!/bin/csh
if ( $?MANPATH == 0 ) then
setenv MANPATH :/usr/local/mpi/man
else
setenv MANPATH ${MANPATH}:/usr/local/mpi/man
endif
setenv PATH ${PATH}:/usr/local/mpi/bin
```

图 3-7　mpich. csh 文件

③ $ cd /etc。

$./profile 让新加的 PATH 立即生效，至此，环境变量的配置完成。

2）在集群上 MPI 环境配置

在集群上对 MPI 环境进行配置的步骤与在单机上大致相同。在集群环境中，由于物理结点不止一个，所以在修改/etc/host 文件时，应填入相应 node 及其 ip 值。

2. 开发第一个 MPI 程序

下面，以一个简单的实例“Hello World”为例，给出 MPI 程序的一个简单框架，使读者对 MPI 并行程序有一个基本的认识。

C 语言的程序设计者对“Hello World”这一例子一定还记忆犹新，几乎每本程序设计书都是从这个简单的例子程序开始的。它虽然简单，但具有一定的代表性。因此，本书也从“Hello World”开始对 MPI 编程进行介绍，其代码如下所示：

```
#include "mpi.h"                              /* 包含 mpi 头文件 */
#include<stdio.h>
#include<math.h>
void main(int argc,char *argv[])
{
         int myid,numprocs;
         int namelen;
         char processor_name[MPI_MAX_PROCESSOR_NAME];
         MPI_Init(&argc,&argv);               /* 初始化 mpi,并行程序开始执行 */
         /* 获得当前进程的标识号 */
```

```
        MPI_Comm_rank(MPI_COMM_WORLD,&myid);
        MPI_Comm_size(MPI_COMM_WORLD,&numprocs);  /* 获得总进程数 */
        MPI_Get_processor_name(processor_name,&namelen);
        /* 返回当前进程所在的机器名 */
        printf("Hello World! Process %d of %d on %s\n",myid,numprocs,processor_name);
        MPI_Finalize();                           /* 结束 mpi */
}
```

对这第一个C+MPI并行程序，分为如下四部分进行讲解：

① 需要MPI相对于C实现的头文件mpi.h。

② 定义程序中所需要的与MPI有关的变量。MPI_MAX_PROCESSOR_NAME是MPI预定义的宏，即在某一MPI的具体实现中允许机器名字的最大长度。机器名放在变量processor_name中。整型变量myid和numprocs分别用来记录并行执行的当前进程的标识和所有参加计算的进程的个数。namelen是实际得到的机器名字的长度。

③ 在MPI程序的开始和结束处，必须分别调用MPI_Init与MPI_Finalize函数，各自完成MPI程序的初始化和结束工作。

④ MPI程序的程序体，包括各种MPI过程调用语句和C语句。MPI_Comm_rank得到当前正在运行的进程的标识号，放在myid中；MPI_Comm_size得到所有参加运算的进程的个数，放在numprocs中；MPI_Get_processor_name得到当前进程所在机器的名称，结果放在processor_name中，它是一个字符串，而该字符串的长度放在namelen中，fprintf语句将当前进程的标识号、并行执行的进程的个数、当前进程所在机器的名字打印出来。

与普通C程序不同的是，在这些程序体中的执行语句是并行执行的，每一个进程都要执行。不妨指定本程序启动时共产生了4个进程同时运行，而运行本程序的机器的名称为localhost，这4个进程都在localhost上运行，其标识分别为0、1、2、3，执行结果如图3-8所示。虽然，该MPI程序本身只有一条打印语句，但是由于它启动了4个进程同时执行，每个进程都要执行这条打印操作，所以最终的执行结果有4条打印语句。

```
Hello World! Process 0 of 4 on localhost
Hello World! Process 3 of 4 on localhost
Hello World! Process 2 of 4 on localhost
Hello World! Process 1 of 4 on localhost
```

图3-8 Hello World程序在单机上的运行结果

运行此程序：

- 在当前文件夹中，建立一个test.c文件，并编写上述代码。
- 编译此文件$mpicc -o test test.c。
- 运行目标文件$mpirun -np 4 test。

-np 4表示生成4个进程并行执行此程序，执行结果如图3-8所示。

该程序的并行执行过程如图 3-9。即 4 个进程所运行的机器是相同的。由于 4 个进程同时执行，在本程序中没有限定打印语句的顺序，因此无论哪个进程的打印语句在前，哪个在后，都是可能的，只要有 4 条正确的输出语句即可。

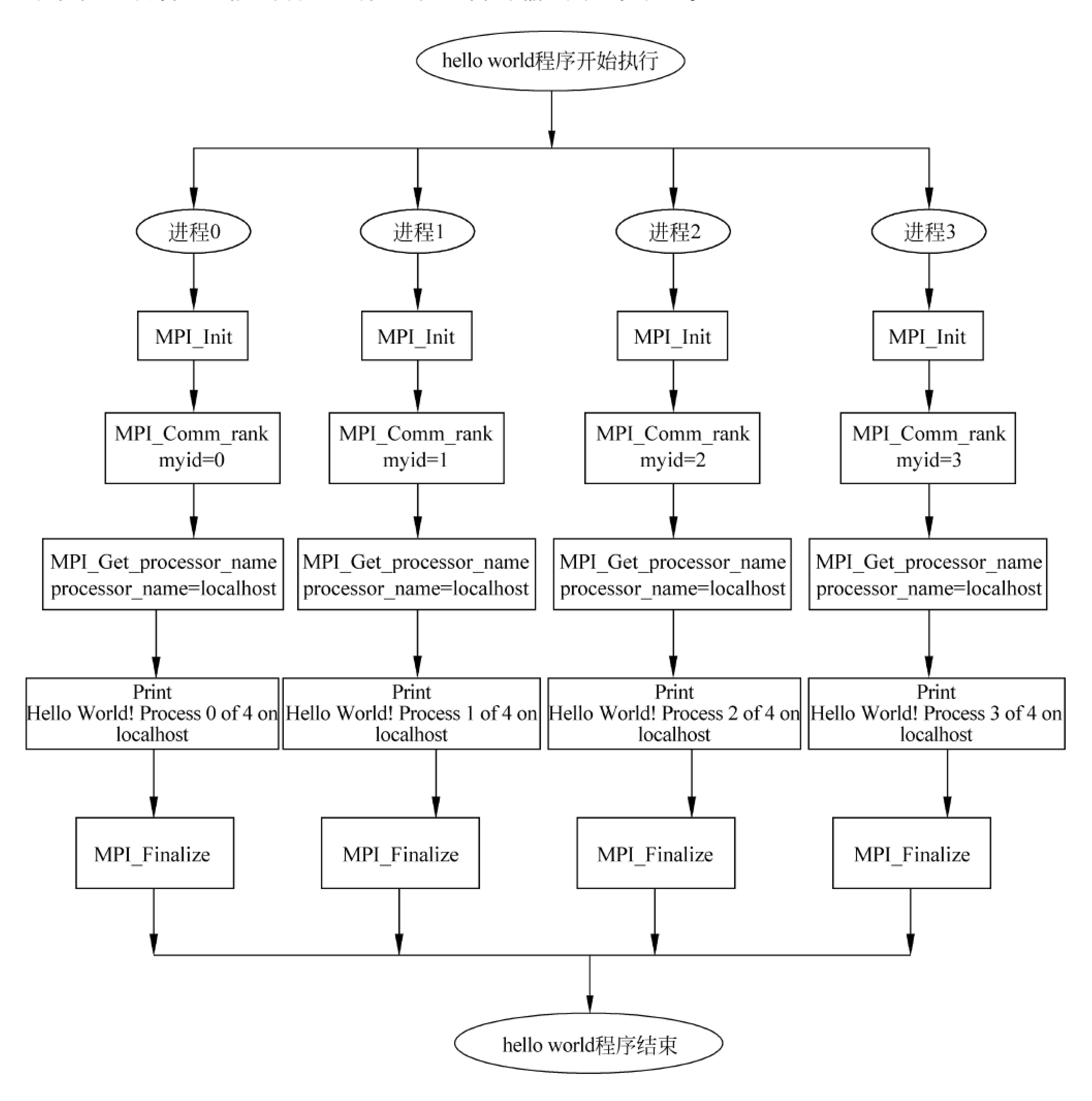


图 3-9　Hello World 程序的运行过程

3. MPI 几个简单函数

1）MPI 参数说明

MPI 常用参数如下：

- 缓冲区(buffer) 指由应用程序定义的用于发送或接收数据的缓冲区。
- 数据个数(count) 指发送或接收指定数据类型的数据个数。
- 数据类型(type) MPI 定义了一些默认的数据类型,用户也可以根据需要创建自己的数据类型。其中,MPI_BYTE 和 MPI_PACKED 与 C 语言的类型不对应。
- 目的地(dest) 发送进程指定地接收该消息的目的进程,也就是接收进程的标识号。
- 源(source) 接收进程指定地发送该消息的源进程,也就是发送进程的标识号。如果该值为 MPI_ANY_SOURCE 表示接收任意源进程发来的消息。
- 标识符(tag) 为了标识一个消息,由程序员指定的唯一非负整数值(0～32 767)。发送操作和接收操作的标识符一定要匹配,但对于接收操作来说,如果 tag 指定为 MPI_ANY_TAG 则可与任何发送操作的 tag 相匹配。
- 通信因子(comm) 包含源进程与目的进程的一组上下文相关的进程集合,除非用户自己定义(创建)了新的通信因子,否则均使用系统预先定义的全局通信因子 MPI_COMM_WORLD。
- 状态(status) 对于接收操作,包含了接收消息的源进程(source)和消息的标识符(tag)。在 C 程序中,这个参数是指向 MPI_Status 结构的指针(如 status. MPI_SOURCE、status. MPI_TAG)。而在 Fortran 程序中,这个参数是大小为 MPI_STATUS_SIZE 的整型数组(如 status(MPI_SOURCE)、status(MPI_TAG))。另外,实际接收到的消息长度可以通过 MPI_Get_count()函数从该参数中得到。
- 请求(request) 这个参数用于非阻塞发送和非阻塞接收操作。由于非阻塞操作返回后,实际上消息并未真正完成发送或接收。因此,用户可以根据该变量调用其他函数完成消息的实际发送和接收。在 C 程序中,这个参数是指向 MPI_Request 结构的指针。

MPI 对参数说明的方式有三种,分别是 IN、OUT 和 INOUT,它们的含义分别是:

- IN(输入) 函数调用传递给 MPI 的参数,MPI 除了使用该参数外不允许对这一参数做任何修改。
- OUT(输出) MPI 返回给函数调用的结果参数,该参数的初始值对 MPI 没有任何意义。
- INOUT(输入输出) 函数调用首先将该参数传递给 MPI,MPI 对这一参数进行引用,修改之后再将结果返回给外部函数调用,该参数的初始值和返回结果都有意义。

2) 常用函数

① MPI_Init( )

MPI_Init( )是初始化 MPI 运行环境的函数。每个 MPI 程序必须调用该函数,并且它必须在所有调用 MPI 函数之前被调用,而且只能被调用一次。对于 C 程序,MPI_Init 必须传递所有的命令行参数。

```
MPI_Init ( * argc,  ** argv)
```

② MPI_Comm_size

该函数返回在当前通信域中所包含的进程数，不同的进程通过这一调用可以得到当前通信域中一共有多少个进程在并行执行。通常，可以根据通信因子 MPI_COMM_WORLD 来查询用户程序包含的进程数。

```
MPI_Comm_size (comm,  * size)
```

③ MPI_Comm_rank

该函数返回当前进程在指定通信域中的进程标识号，不同的进程可以依据进程号将自身和其他进程区别开来，从而实现各进程的并行执行和协作。一个进程在不同通信域中的进程标识号可能不同。

```
MPI_Comm_rank (comm,  * rank)
```

④ MPI_Finalize

该函数结束 MPI 执行环境。该函数一旦被应用程序所调用，就不能再调用 MPI 的其他函数(包括 MPI_Init)。用户必须保证在进程调用 MPI_Finalize 之前结束与进程有关的所有通信结束。

```
MPI_Finalize ()
```

⑤ MPI_Send

该函数是最基本的阻塞发送函数。当函数返回时，应用程序的发送缓冲区空闲，可以继续使用。

```
MPI_Send ( * buf, count, datatype, dest, tag, comm)
```

⑥ MPI_Recv

该函数是最基本的阻塞接收函数，直到接收的消息到达本进程的接收缓冲区之后该函数调用才返回。

```
MPI_Recv ( * buf, count, datatype, source, tag, comm,  * status)
```

⑦ MPI_Abort

结束所有与该通信域相关的进程。一般来说，调用该函数后，所有的进程都退出，不管该进程是否与该通信域相关。

```
MPI_Abort (comm, errorcode)
```

⑧ MPI_Get_processor_name

该函数返回当前进程所在处理器的名称，name 缓冲区的大小必须大于 MPI_MAX_

PROCESSOR_NAME，真正的长度返回在 resultlength 变量中。

```
MPI_Get_processor_name ( * name,  * resultlength)
```

⑨ MPI_Wtime

该函数返回调用进程已执行的时间(以秒为单位，双精度)。

```
MPI_Wtime ()
```

⑩ MPI_Wtick

该函数按秒返回 MPI_Wtime 的分辨率，也就是返回一个双精度值(连续时间之间的秒数)。例如，如果时钟由作为按毫秒递增的计数器来实现时，则 MPI_Wtick 返回的值是 10^{-3}。

3.1.3 点对点通信

在 MPI 系统中，完成一次消息传递至少需要两个以上的进程，且至少有一个消息发送进程和至少一个消息接收进程。点对点通信就是 MPI 程序中任意两个进程之间的一次消息传递。其中一个进程负责发送消息，另一个进程负责接收消息。称发送消息的进程为发送进程，接收消息的进程为接收进程，并且在一次点对点通信中发送进程和接收进程必须是相互匹配的。

1. 点对点通信的分类

点对点通信根据消息发送和接收方式的不同可以分为阻塞通信和非阻塞通信两大类。其中阻塞通信和非阻塞通信又分别分为标准模式、缓存模式、就绪模式和同步模式。具体分类如表 3-1 所示。

表 3-1 点对点通信模型分类

<table>
<tr><td colspan="2" rowspan="4">阻塞通信</td><td>标准通信模式</td><td>MPI_Send</td><td rowspan="4">MPI_Recv
MPI_Irecv
MPI_Recv_Init</td></tr>
<tr><td>缓存通信模式</td><td>MPI_Bsend</td></tr>
<tr><td>就绪通信模式</td><td>MPI_Rsend</td></tr>
<tr><td>同步通信模式</td><td>MPI_Ssend</td></tr>
<tr><td rowspan="8">非阻塞通信</td><td rowspan="4">非重复</td><td>标准通信模式</td><td>MPI_ISend</td><td rowspan="8">MPI_Recv
MPI_Irecv
MPI_Recv_Init</td></tr>
<tr><td>缓冲通信模式</td><td>MPI_IBsend</td></tr>
<tr><td>就绪通信模式</td><td>MPI_IRsend</td></tr>
<tr><td>同步通信模式</td><td>MPI_ISsend</td></tr>
<tr><td rowspan="4">重复</td><td>标准通信模式</td><td>MPI_Isend_Init</td></tr>
<tr><td>缓冲通信模式</td><td>MPI_Ibsend_Init</td></tr>
<tr><td>就绪通信模式</td><td>MPI_Irsend_Init</td></tr>
<tr><td>同步通信模式</td><td>MPI_Issend_Init</td></tr>
</table>

2. 阻塞通信

阻塞通信是有序的，即接收进程要按照发送进程的发送顺序来接收，即使后发送的消息先到达。此外，在阻塞通信模型中，当一个通信操作正确返回时，则该通信操作已正确完成(消息成功发送或接收)，该操作所占用的缓冲区可被其他操作继续使用。阻塞通信模型的工作流程图如图 3-10 所示。

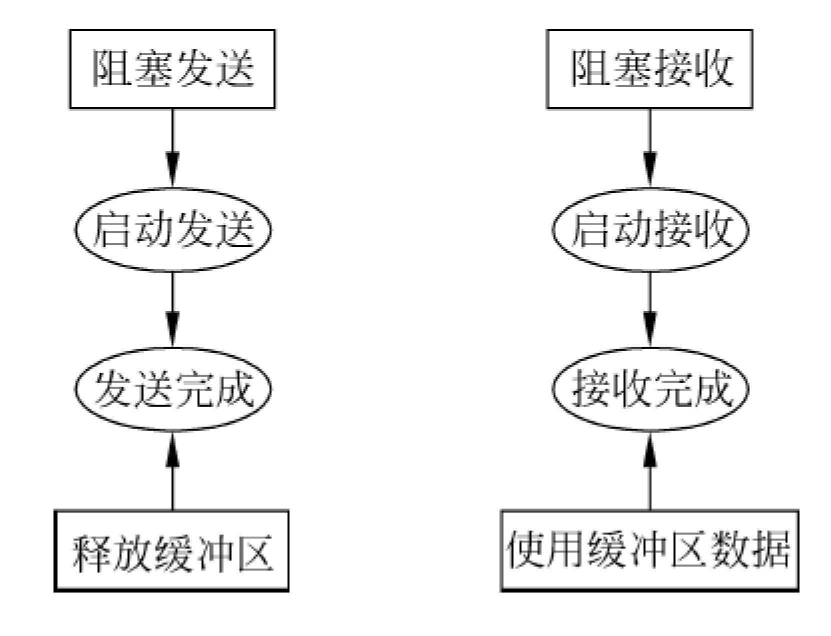


图 3-10　阻塞通信模型的流程图

阻塞通信需要消息发送进程的发送操作与接收进程的接收操作相互配合来完成。在调用发送函数返回之前，需要将要发送的消息安全地托管，即下次发送函数的调用不会影响上次发送的数据。按照发送的消息被托管方式的不同，阻塞通信又可被分为四种模式，在不同模式下要求 MPI 环境本身提供的缓存机制各有不同。这四种通信模式的分类如表 3-2 所示。

表 3-2　阻塞通信的四种通信模式

通 信 模 式	函 数 原 型
标准通信模式	MPI_Send (* buf, count, datatype, dest, tag, comm)
缓存通信模式	MPI_Bsend (* buf, count, datatype, dest, tag, comm)
就绪通信模式	MPI_Rsend (* buf, count, datatype, dest, tag, comm)
同步通信模式	MPI_Ssend (* buf, count, datatype, dest, tag, comm)

1) 标准通信模式

标准通信模式是最基本的阻塞通信模式。当此模式中，当发送函数返回时，表明应用程序的发送缓冲区空闲，可以被下次发送函数继续使用。其函数的原型如下：

```
MPI_Send(void * buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
IN          buf          发送缓冲区的起始地址
IN          count        发送数据的个数
IN          datatype     发送数据的数据类型
IN          dest         目的进程的标识号
IN          tag          消息标签
IN          comm         通信域
```

在这种模式中，由 MPI 决定是否需要缓存正要发出的消息。当 MPI 决定并缓存这些消息时，则发送函数可以直接返回而无须等待对应接收进程启动接收操作。若 MPI 缓存空间不足或因为其他原因，MPI 选择不缓存正要发出的消息。这时，发送操作必须等

待相应的接收操作启动并且数据被接收者接收后才可以返回。所以，标准模式发送操作是否开始并不依赖于匹配的接收操作是否启动，它可以早于匹配接收启动进行。但标准模式的发送又是全局的操作，发送操作的完成依赖于匹配接收操作的状态。

缓存可以提高一个正确程序的性能，但它不影响程序运行的结果。在阻塞标准通信模式中，MPI 没有规定发送进程要发送的消息是否被缓存，这使得程序不依赖于系统缓存，从而提高了程序的可移植性。

阻塞标准通信模式的工作流程如图 3-11 所示。

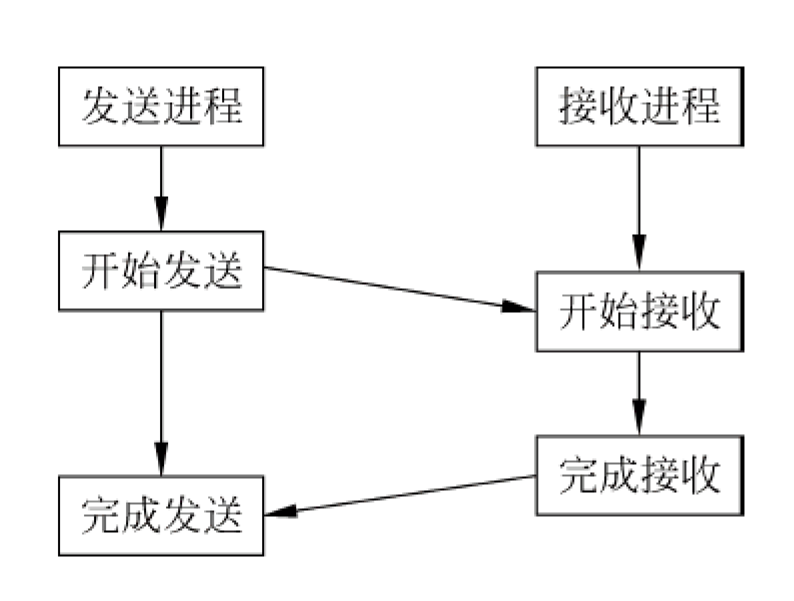


图 3-11　阻塞标准通信模式的流程图

2）缓存通信模式

与标准通信模式不同的是，在缓存通信模式中，用户可以直接对通信缓冲区进行申请、使用和释放，对缓冲区的设计利用完全由程序员决定。在缓存通信模式中无论匹配进程的接收操作是否启动，发送操作都可以执行之后就返回。但是，在发送消息之前必须保证有缓冲区可用，否则该发送将直接错误返回。

在缓存通信模式中，消息发送能否进行以及能否正确返回并不依赖于接收进程，而是完全依赖于是否有足够的通信缓冲区可用。当有足够的缓冲区时，发送消息被缓存之后，发送操作即可成功返回。但若没有足够的缓冲区，则发送操作直接错误返回。缓存发送返回后，并不意味着该缓冲区可以自由使用，只有当缓冲区中的消息被发送出去之后，才可以释放该缓冲区。其函数的原型如下：

```
MPI_Bsend (void * buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

（具体的参数含义见标准通信模式）

在进行通信时，用户需要根据需求向系统申请一定大小的缓冲区，申请成功之后就可以利用这些缓冲区对发送的消息进行缓存。在让消息接收完毕之后，就可以释放这些缓冲区。申请和释放缓冲区的函数如表 3-3 所示。

表 3-3　申请和释放缓冲区函数

申请函数	MPI_Buffer_attach (* buffer, size)
释放函数	MPI_Buffer_detach (* buffer, size)

其中 buffer 是缓冲区的初始地址，size 是以字节为单位的缓冲区大小。MPI_Buffer_attach 将大小为 size 的缓冲区交给 MPI，用户可以用其来缓冲将要发送的消息。MPI_Buffer_detach 将大小为 size 的缓冲区收回。该操作是阻塞调用，它一直等到使用该缓冲区的消息发送完成之后才返回。MPI_Buffer_detach 释放的缓冲区 buffer 可以被用户重

新使用。

阻塞缓存通信模式的工作流程如图 3-12 所示。

3）就绪通信模式

与标准通信模式不同的是，在就绪通信模式中，只有在匹配进程的接收操作启动之后，发送进程的发送操作才能启动，否则将会出现发送出错。其函数的原型如下：

```
MPI_Rsend(void * buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

（具体的参数含义见标准通信模型）

阻塞就绪通信模式的工作流程如图 3-13 所示。

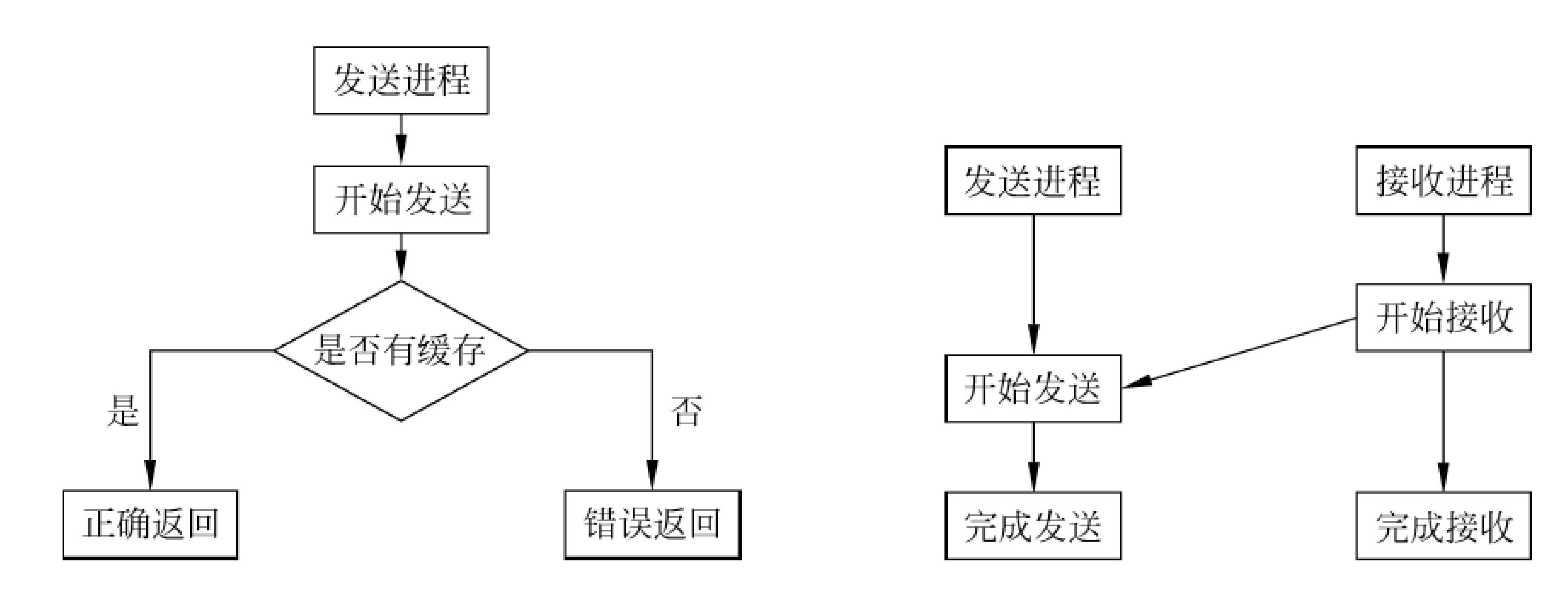


图 3-12　阻塞缓存通信模式的流程图　　图 3-13　阻塞就绪通信模式的流程图

4）同步通信模式

与标准通信模式不同的是，在同步通信模式中，发送操作只要等到相应的接收进程开始之后就可以正确地返回，而此时发送缓冲区的内容也全部被系统缓冲区缓存。所以，当发送正确返回时，发送缓冲区可以被重新使用。其函数的原型如下。

```
MPI_Ssend (void * buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

（具体的参数含义详见标准通信模型）

阻塞同步通信模式的工作流程如图 3-14 所示。

5）程序示例

下面，举一个简单的程序实例，对以上四种阻塞通信模式进行简单的使用。此程序启动 8 个进程。其中，进程 0 以标准模式向进程 4 发送整型消息 0，进程 1 以缓存模式向进程 5 发送整型消息 1，进程 2 以就绪模式向进程 6 发送整型消息 2，进程 3 以同步模式向进程 7 发送整型消息 3。具体的代码如下。

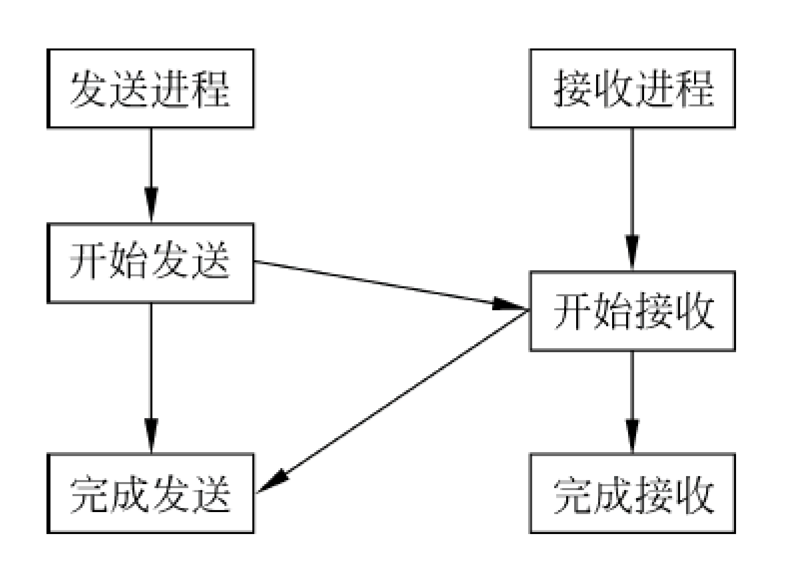


图 3-14　阻塞同步通信模式的流程图

```
int rank,bsize, * buf,recv;
int nums[4] = {0,1,2,3};
MPI_Init(&argc,&argv);          /* 初始化 mpi,并行程序开始执行 */
MPI_Comm_rank(comm,&rank);      /* 获得当前进程的标识号 */
if(rank == 0)                   /* 以标准模式向 4 号进程发送数据 */
    MPI_Send(&nums[rank],1,MPI_INT,4,0,comm);
else if(rank == 1)              /* 以缓存模式向 5 号进程发送数据 */
{
    /* 用 PACK_SIZE 来计算包装一个消息所需缓冲区的大小的上界,以字节为单位 */
    MPI_Pack_size(1,MPI_INT,comm,&bsize);
        /* MPI_BSEND_OBERHEAD 定义使用缓冲区方式所需额外开销的上界 */
        bsize += 2 * MPI_BSEND_OVERHEAD;
    buf = (int *)malloc(bsize);
    /* 装配一个用于通信的缓冲区 */
      MPI_Buffer_attach(buf,bsize + MPI_BSEND_OVERHEAD);
      MPI_Bsend(&nums[rank],1,MPI_INT,5,0,comm);
      /* 卸载用于通信的缓冲区 */
    MPI_Buffer_detach(&buf,&bsize);
    free(buf);
}
else if(rank == 2)              /* 以就绪模式向 6 号进程发送数据 */
    MPI_Rsend(&nums[rank],1,MPI_INT,6,0,comm);
else if(rank == 3)              /* 以同步模式向 7 号进程发送数据 */
    MPI_Ssend(&nums[rank],1,MPI_INT,7,0,comm);
else if(rank == 4)              /* 接收 0 号进程发来的数据 */
{
    MPI_Recv(&recv,1,MPI_INT,0,0,comm,&status);
    printf("Process %d Recv Mes from process %d: %d\n",rank,status.MPI_SOURCE,recv);
}
else if(rank == 5)              /* 接收 1 号进程发来的数据 */
{
    MPI_Recv(&recv,1,MPI_INT,1,0,comm,&status);
    printf("Process %d Recv Mes from process %d: %d\n",rank,status.MPI_SOURCE,recv);
}
else if(rank == 6)              /* 接收 2 号进程发来的数据 */
{
    MPI_Recv(&recv,1,MPI_INT,2,0,comm,&status);
    printf("Process %d Recv Mes from process %d: %d\n",rank,status.MPI_SOURCE,recv);
}
else if(rank == 7)              /* 接收 3 号进程发来的数据 */
{
    MPI_Recv(&recv,1,MPI_INT,3,0,comm,&status);
    printf("Process %d Recv Mes from process %d: %d\n",rank,status.MPI_SOURCE,recv);
}
```

程序运行结果如下所示：

```
Process 4 Recv Mes from process 0: 0
Process 6 Recv Mes from process 2: 2
Process 5 Recv Mes from process 1: 1
Process 7 Recv Mes from process 3: 3
```

3. 非阻塞通信

在阻塞发送模型中，操作完成之前处理器只能等待，这样浪费了处理机的大量资源。为了解决这个问题，可以采用计算和通信重叠的技术，非阻塞通信模型可实现该目的。

在非阻塞通信模型中，通信操作不必等待实际完成便可直接返回。它只对这一通信操作进行了初始化，然后便将该操作交给特定的硬件去完成。与此同时，处理器可以进行其他计算操作，从而实现了计算和通信的重叠。这样的并行设计思路大大提高了程序的执行效率。

阻塞通信模型发送操作的返回意味着发送操作的完成，即发送缓冲区可以被重新使用。而非阻塞通信模型则与之不同，在该模型中，发送操作的完成需要特定的方法，需要检测消息是否发送完成。接收操作亦如此，非阻塞接收的返回并不意味着接收的消息已经全部到达，同样，也需要特定的方法进行检测。非阻塞通信模型的工作流程如图 3-15 所示。

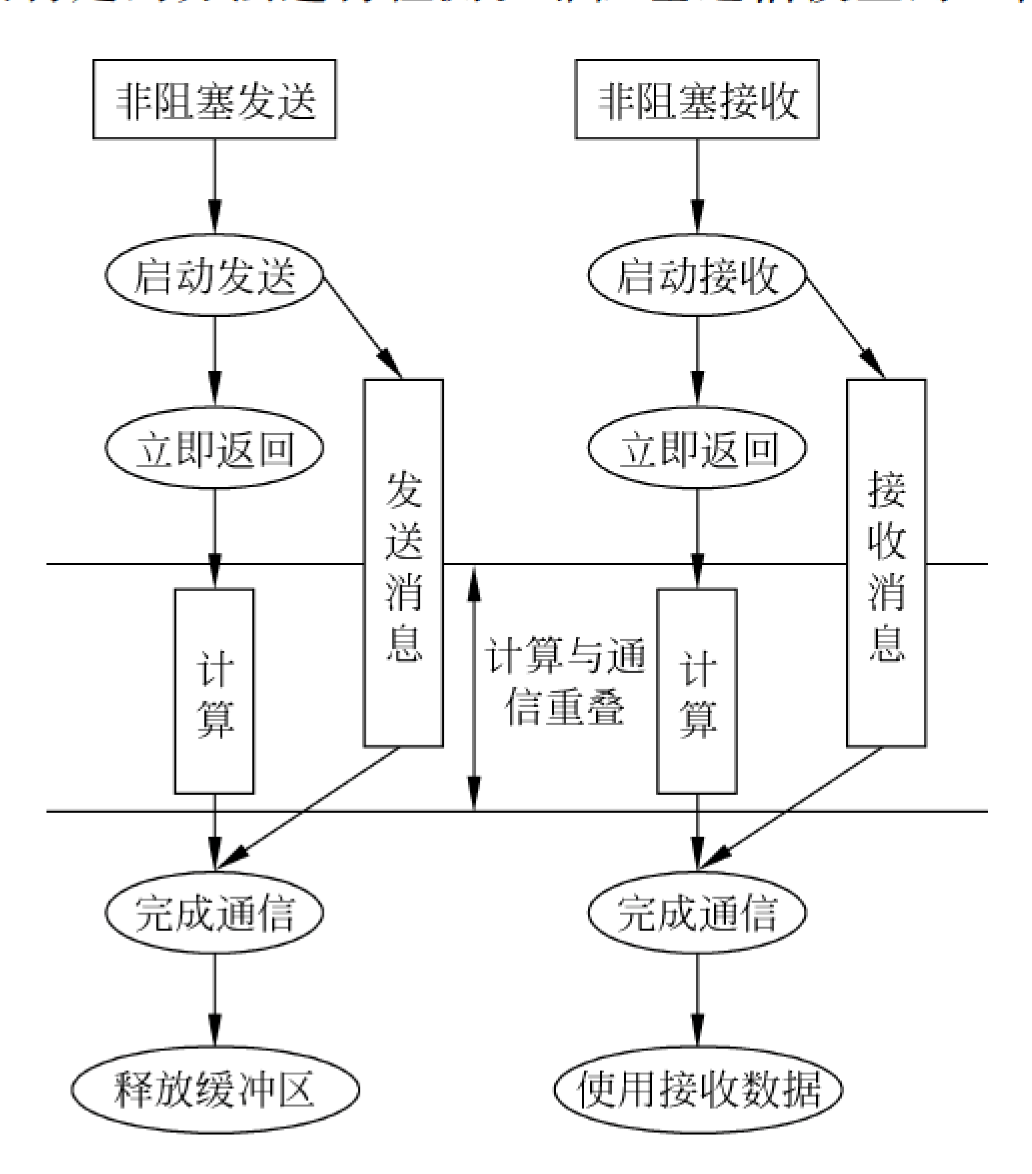


图 3-15　非阻塞通信模型的流程图

1）非阻塞通信模型的分类

相对于阻塞通信包含的四种通信模式，非阻塞通信也可分为相应的四种通信模式，具

体分类情况如表 3-4 所示。

表 3-4　非阻塞通信模型的分类

通信模式	函数原型
标准通信模式	MPI_Isend (* buf，count，datatype，dest，tag，comm)
缓冲通信模式	MPI_Ibsend (* buf，count，datatype，dest，tag，comm)
就绪通信模式	MPI_Irsend (* buf，count，datatype，dest，tag，comm)
同步通信模式	MPI_Issend (* buf，count，datatype，dest，tag，comm)

下面仅对标准通信模式进行详细介绍，非阻塞通信模型中其他模式与标准模式的区别与阻塞通信模型中各模式的区别相似，在此不再进行详细介绍。

2）常用的非阻塞操作

由于非阻塞通信与阻塞通信不同，非阻塞通信操作调用之后立即返回。所以，非阻塞操作除了基本的发送和接收操作之外，还需要其他操作（如状态检测、消息探测等）来配合非阻塞通信操作的完成。在非阻塞通信模型的标准通信模式中，常用的函数如表 3-5 所示。

表 3-5　常用的标准非阻塞通信操作

操作名	函数名
发送操作	MPI_Isend(* buf，count，datatype，dest，tag，comm)
接收操作	MPI_Irecv(* buf，count，datatype，source，tag，comm， * request)
等待完成操作	MPI_Wait(* request， * status)
检测完成操作	MPI_Test(* request， * flag， * status)
取消操作	MPI_Cancel(* request)
检测取消完成操作	MPI_Test_cancelled(* status， * flag)
通信探测操作(阻塞式)	MPI_Probe(source，tag，comm， * status)
通信探测操作(非阻塞式)	MPI_Iprobe(source，tag，comm， * flag， * status)
通信对象释放操作	MPI_Request_free(* request)

(1）发送函数

函数原型如下：

```
int MPI_Isend(void * buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm
            comm, MPI_Request * request)
    IN          buf             发送缓冲区的起始地址
    IN          count           发送数据的个数
    IN          datatype        发送数据的数据类型
    IN          dest            目的进程的标识号
    IN          tag             消息标签
```

```
    IN          comm          通信域
    OUT         request       非阻塞通信对象
```

发送操作被调用之后立即返回,但它只是对发送操作进行了初始化,并不代表消息发送的完成。与标准阻塞发送函数相比,该函数多了一个 request 参数。它是一个用来描述该非阻塞通信状态的对象,通过对 request 相对应的通信状态进行查询,可以判定本次非阻塞发送操作是否完成。

(2) 接收函数

函数原型如下:

```
int MPI_Irecv(void * buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm
              comm, MPI_Request * request)
    OUT         buf           接收缓冲区的起始地址
    IN          count         接收数据的个数
    IN          datatype      接收数据的数据类型
    IN          source        源进程的标识号
    IN          tag           消息标签
    IN          comm          通信域
    OUT         request       非阻塞通信对象
```

接收操作被调用之后立即返回,但并不意味着全部的消息已接收完成。函数中的 request 参数是一个用来描述本次通信状态的对象,通过对 request 相对应的通信状态进行查询,可以得知本次非阻塞接收操作是否完成。

(3) 等待完成操作

函数原型如下:

```
int MPI_Wait(MPI_Request * request, MPI_Status * status)
    INOUT       request       非阻塞通信对象
    OUT         status        通信状态
```

MPI_Wait 操作要一直等到与该非阻塞通信对象相对应的本次非阻塞通信操作(如发送操作、接收操作)完成之后才返回,同时释放该阻塞通信对象。函数中的 Status 参数存储了非阻塞通信的完成状态。

(4) 检测完成操作

函数原型如下:

```
int MPI_Test(MPI_Request * request, int * flag, MPI_Status * status)
    INOUT       request       非阻塞通信对象
    OUT         flag          操作是否完成标志
    OUT         status        通信状态
```

MPI_Test 操作也是以 request 为参数。但与 MPI_Wait 不同的是,它不必等到与此

非阻塞通信对象相对应的通信操作完成之后才返回。在执行 MPI_Test 操作时，如果要查询的非阻塞通信操作已经完成，则它完成于 MPI_Wait 相同的功能，成功返回，置 flag 为 true，并释放 request 对象；否则，它不必等待非阻塞通信操作结束而立即返回，并置 flag 为 false。同时也无须释放 request 对象。

(5) 取消操作

函数原型如下：

```
int MPI_Cancel(MPI_Request * request)
    OUTIN          request          非阻塞通信对象
```

MPI_Cancel 以 request 为参数，可以取消与该通信对象相关的非阻塞操作，并释放该操作所占用的资源。该操作可以立即返回，但是却不能保证取消操作执行成功。取消操作被调用时，如果相对应的非阻塞通信已经开始，则该操作会正常完成，不受取消操作的影响。若取消操作调用时，如果相对应的非阻塞通信还没有开始，则可以释放通信所占用的资源，取消该非阻塞通信。对于非阻塞通信，即使调用了取消操作，也必须调用非阻塞通信的完成操作或查询对象的释放操作来释放通信对象。

(6) 检测取消完成操作

函数原型如下：

```
int MPI_Test_cancelled(MPI_Status * status, int * flag)
    OUT            status           通信状态
    OUT            flag             是否取消标志
```

MPI_Test_cancelled 操作以 request 为参数，可以检查与该 request 相关的非阻塞通信操作是否被取消，并将返回结果写入 flag 中。如果 flag 为 true，则该通信操作已被取消；否则，该通信操作没有被取消。

(7) 通信探测操作

阻塞通信探测函数原型如下：

```
MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status * status)
    IN             source           进程的标识号或 MPI_ANY_SOURCE(任意源)
    IN             tag              消息标签或 MPI_ANY_TAG(任意标签)
    IN             comm             通信域
    OUT            status           通信状态
```

非阻塞通信探测函数原型如下：

```
MPI_Iprobe(int source, int tag, MPI_Comm comm, int * flag, MPI_Status * status)
    IN             source           进程的标识号或 MPI_ANY_SOURCE(任意源)
    IN             tag              消息标签或 MPI_ANY_TAG(任意标签)
    IN             comm             通信域
```

```
OUT          flag          是否有消息到达的标志
OUT          status        通信状态
```

MPI_Probe 和 MPI_Iprobe 操作允许在没有实际执行消息接收的情况下对其进行检查。当存在一个消息可被接收并且 MPI_Iprobe 与参数 source、tag、comm 相匹配时，它返回的 flag 值为 true。该调用返回的状态 status 与执行接收操作所收到的消息状态相同，可以从返回的状态 status 中获取 source、tag 和检查消息的长度。然后，可以使用相匹配的接收操作接收该消息。

MPI_Iprobe 的 source 参数可以是 MPI_ANY_SOURCE(任意源)，tag 参数可以是 MPI_ANY_TAG(任意标签)，以使用户可以检查来自不确定的源和不确定标签的消息。不过，该操作必须指定一个通信域 comm 来指明通信的上下文。一个消息被检查后并不一定被立即接收，同样，一个消息在被接收之前可能被检查多次。MPI_Probe 与 MPI_Iprobe 相似，只是它是一个阻塞调用，只有找到一个匹配调用之后才返回。

(8) 通信对象释放操作

函数原型如下：

```
int MPI_Request_free(MPI_Request * request)
    INPUT          request          非阻塞通信对象
```

如果确定一个非阻塞操作已经完成，则可以直接调用非阻塞通信对象释放语句 MPI_Request_free 将该对象所占用的资源释放，而不是通过调用非阻塞通信完成操作来间接进行释放。一旦执行了释放操作，非阻塞通信对象就无法再通过其他任何的调用访问。但是，如果与该非阻塞通信对象相关的通信还没有完成，则该对象的资源并不会立即释放，它将等到该非阻塞通信结束之后再释放，因此，非阻塞通信对象的释放并不影响该非阻塞通信的完成。

3) 程序实例

针对上面介绍的函数，使用如下程序实例进行简单示范。该程序开启了三个进程，进程 0 采用非阻塞标准通信模式分别向进程 1 和进程 2 发送一个整型数据。然后，进程 1 用 MPI_Probe 方法探测来自任意进程、任意标签的消息，如果检测到是进程 0 发来的消息则采用非阻塞方式接收，并采用 MPI_Test 方法检测接收操作是否完成。对于进程 2，则先采用非阻塞方式接收，然后采用 MPI_Cancel 方法取消本次接收操作，并用 MPI_Test_cancelled 方法判断取消操作是否成功。具体的程序代码如下。

```
MPI_Request request[2];
int rank,bsize, * buf,recv;
int nums[2] = {0,1};
MPI_Init(&argc,&argv);          /* 初始化 mpi,并行程序开始执行 */
MPI_Comm_rank(comm,&rank);      /* 获得当前进程的标识号 */
```

```
if(rank == 0)
{
    int i;
    /* 以非阻塞的方式向进程 1 和进程 2 发送消息 */
    for(i = 0;i < 2;i++)
    {
        MPI_Isend(&nums[i],1,MPI_INT,i + 1,0,comm,&request[i]);
        printf("Process 0 start the Isend Op to process %d.\n",i + 1);
    }
    for(i = 0;i < 2;i++)
    {
        /* 等待消息发送操作的完成 */
        MPI_Wait(&request[i],&status);
        printf("Process 0 have sent Mes to process %d.\n",i + 1);
    }
}
else if(rank == 1)
{
    /* 探测来自任意源、带有任意标签的消息
     * MPI_ANY_SOURCE 任意源
     * MPI_ANY_TAG 任意标签 */
    MPI_Probe(MPI_ANY_SOURCE,MPI_ANY_TAG,comm,&status);
    if(status.MPI_SOURCE == 0 && status.MPI_TAG == 0)
    {
        /* 以非阻塞方式接收来自进程 0 发来的消息 */
        MPI_Irecv(&recv,1,MPI_INT,0,0,comm,&request[0]);
        printf("Process 1 start the Irecv Op from process 0.\n");
    }
    int flag = 0;
    /* 检测接收操作是否完成,并立即返回 */
    MPI_Test(&request[0],&flag,&status);
    if(flag)                    /* 消息接收操作完成 */
        printf("Process 1 Recv Mes From Process 0: %d\n",recv);
}
else if(rank == 2)
{
    MPI_Irecv(&recv,1,MPI_INT,0,0,comm,&request[1]);
    printf("Process 2 start the Irecv Op from process 0.\n");
    /* 取消此次非阻塞接收操作.即使调用了取消操作,也必须调用非阻塞通信的完
     * 成操作或查询对象的释放操作来释放查询对象 */
    MPI_Cancel(&request[1]);
    MPI_Wait(&request[1],&status);
    int flag = 0;
    /* 检测取消操作是否成功 */
```

```
    MPI_Test_cancelled(&status,&flag);
    if(flag)                        /* 取消操作成功 */
        printf("Process 2 have cancelled the Irecv Op from Process 0.\n");
}
```

程序运行结果如下所示：

```
Process 0 start the Isend Op to process 1.
Process 0 start the Isend Op to process 2.
Process 0 have sent Mes to process 1.
Process 0 have sent Mes to process 2.
Process 1 start the Irecv Op from process 0.
Process 1 Recv Mes From Process 0:0
Process 2 start the Irecv Op from process 0.
Process 2 have cancelled the Irecv Op from Process 0.
```

3.1.4 集合通信

集合通信需要一个通信域内所有的进程都要参加。如果把点对点通信比作两个人之间打电话，那么集合通信则相当于小组内所有成员一起讨论。集合通信包括很多类型，按照通信的方向可将集合通信分为：

- 一对多通信；
- 多对一通信；
- 多对多通信。

按通信方式，又可将集合通信分为：

- 同步　通信域中所有进程都到达之后，每个进程再继续运行；
- 数据传递　广播、分散、收集、全部到全部；
- 规约　通信域中的其中一个进程收集所有进程的数据并计算（如求最大值、求最小值、加、乘等）。

1. 一对多集合通信函数

一对多通信，即在一个通信域内，以其中一个进程为根进程，向其他所有的进程（包括该进程自己）发送数据。一对多通信包括 MPI_Bcast（广播）和 MPI_Scatter（散发）两种操作方式。

1）MPI_Bcast

函数原型如下：

```
int MPI_Bcast(void * buffer ,int count, MPI_Datatype datatype, int root, MPI_Comm comm)
    INOUT       buffer          通信消息缓冲区的起始地址
    IN          count           即将广播出去/或接收的数据个数
    IN          datatype        广播/接收数据的数据类型
    IN          root            根进程的标识号
```

IN comm 通信域

在 MPI_Bcast 操作中，给定的根(root)进程将一条消息广播发送到通信域内的所有其他的进程，也包括该进程本身在内。在执行该调用时，通信域内所有进程(包括该进程自己)都使用同一个通信域 comm 和根标识 root，其执行结果是将根进程通信的缓冲区中的消息拷贝到其他所有的进程中去。在执行广播操作时，其他进程指定的通信元素个数(count)、数据类型(datatype)必须与根进程指定的通信元素个数(count)、数据类型(datatype)保持一致。即对于广播操作，不管是广播消息的根进程，还是从根进程接收消息的其他进程，在调用形式上完全一致，都需指明相同的根进程、相同的元素个数以及相同的数据类型。除 MPI_Bcast 之外，其他集合通信都有此限制。MPI_Bcast 的工作示意图如图 3-16 所示。

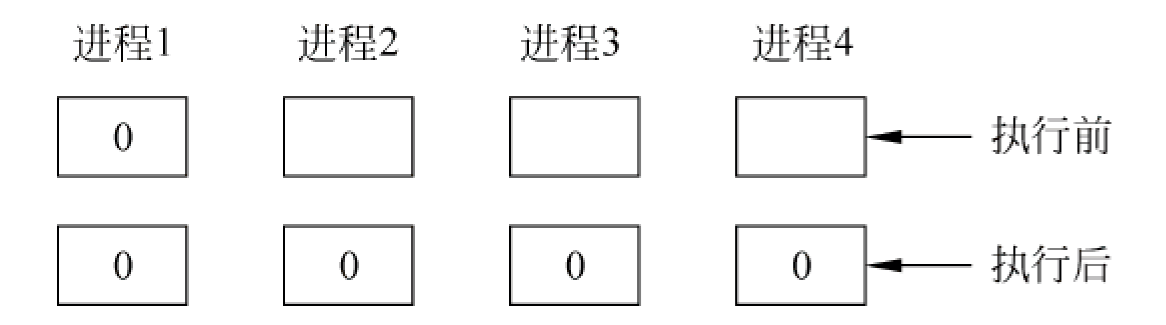


图 3-16 MPI_Bcast 通信的示意图

2) MPI_Scatter

函数原型如下：

```
int MPI_Scatter(void * sendbuf, int sendcount, MPI_Datatype sendtype, void * recvbuf, int
                recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

IN	sendbuf	发送消息缓冲区的起始地址
IN	sendcount	发送到各个进程的数据个数
IN	sendtype	发送消息缓冲区中的数据类型
OUT	recvbuf	接收消息缓冲区的起始地址
IN	recvcount	待接收的元素个数
IN	recvtype	接收元素的数据类型
IN	root	根进程的标识号
IN	comm	通信域

与 MPI_Bcast 不同的是，MPI_Scatter 给定的根(root)进程向其他进程发送的数据可以是不同的。对于所有的非根进程，发送消息缓冲区被忽略。根进程中的发送数据元素个数 sendcount 和发送数据类型 sendtype 必须与所有进程的接收数据元素个数 recvcount 和接收数据类型 recvtype 相同。根进程发送数据元素个数指的是发送给每一个进程的数据元素的个数，而不是总的数据个数。这就意味着在每个进程域根进程之间，发送的数据个数必须和接收的数据个数相等。MPI_Scatter 的工作示意图如图 3-17 所示。

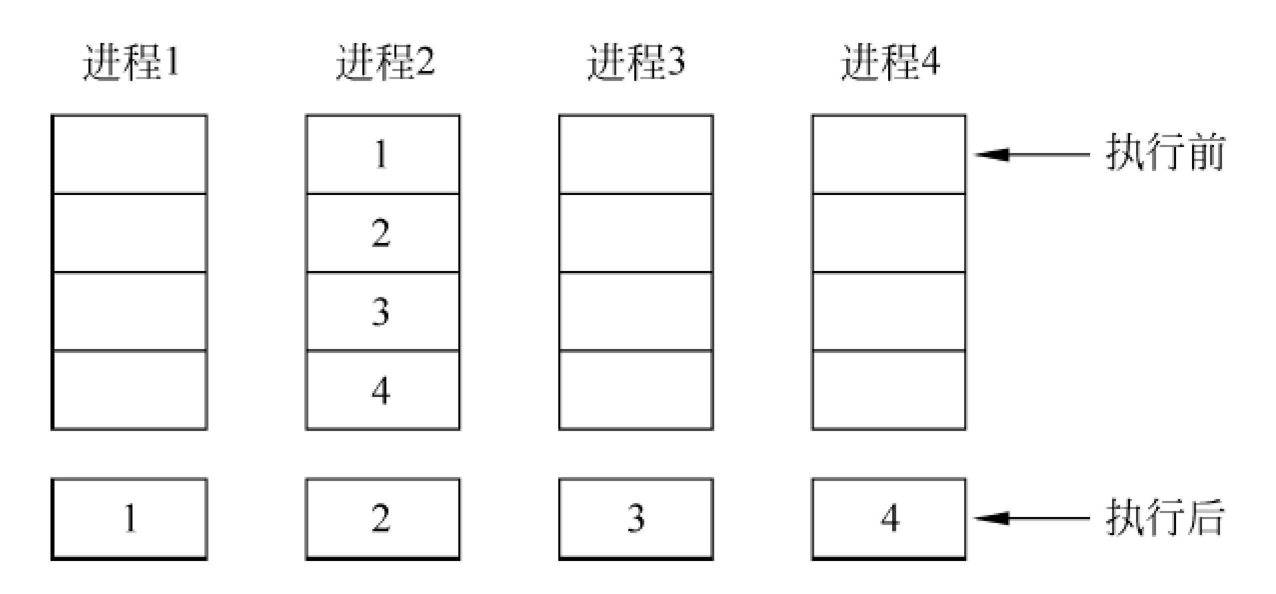


图 3-17　MPI_Scatter 通信的示意图

3）程序示例

```
int rank,value;
MPI_Init(&argc,&argv);          /* 初始化 mpi,并行程序开始执行 */
MPI_Comm_rank(comm,&rank);      /* 获得进程号 */
if(rank == 0)
    value = 0;
/* 进程 0 将 value = 0 广播出去 */
MPI_Bcast(&value,1,MPI_INT,0,comm);
if(rank == 0)
    printf("Process 0 execute bcast:value = 0\n");
printf("Process %d Recv bcast Mes:value = %d\n",rank,value);
```

程序运行结果如下所示：

```
Process 0 execute bcast:value=0
Process 0 Recv bcast Mes:value=0
Process 2 Recv bcast Mes:value=0
Process 1 Recv bcast Mes:value=0
Process 3 Recv bcast Mes:value=0
```

2. 多对一集合通信函数

多对一通信，即在一个通信域内，以其中一个进程为根进程，其他所有的进程（包括该进程自己）向根进程发送数据。多对一通信包括 MPI_Gather（收集）和 MPI_Reduce（规约）两种操作方式。

1）MPI_Gather

函数原型如下：

```
int MPI_Gather(void * sendbuf, int sendcount, MPI_Datatype sendtype, void * recvbuf, int
            recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
    IN        sendbuf          发送消息缓冲区的起始地址
    IN        sendcount        发送消息缓冲区中的数据个数
    IN        sendtype         发送消息缓冲区中的数据类型
```

```
    OUT         recvbuf             接收消息缓冲区的起始地址
    IN          recvcount           待接收的元素个数
    IN          recvtype            接收元素的数据类型
    IN          root                根进程的标识号
    IN          comm                通信域
```

MPI_Gather 是 MPI_Scatter 的逆过程。在收集操作中，每个进程(包括根进程本身)将其发送缓冲区中的消息发送到给定的根(root)进程，根进程根据发送进程的进程标识号，将它们各自的消息依次存放到自己的消息缓冲区中。与广播操作(广播出去的数据都是相同的)不同的是，收集操作从各个进程收集到的数据一般是互不相同的。其结果就像一个进程组中的 N 个进程(包括根进程在内)都执行一个发送调用，而根进程执行了 N 次接收调用。

在收集调用中，每个进程的发送数据个数 sendcount、发送数据类型 sendtype 都是相同的，均分别与根进程中接收数据个数 recvcount、接收数据类型 recvtype 相同。注意根进程中指定的接收数据个数是指从每一个进程接收数据的个数，而不是总的接收个数。MPI_Gather 的工作示意图如图 3-18 所示。

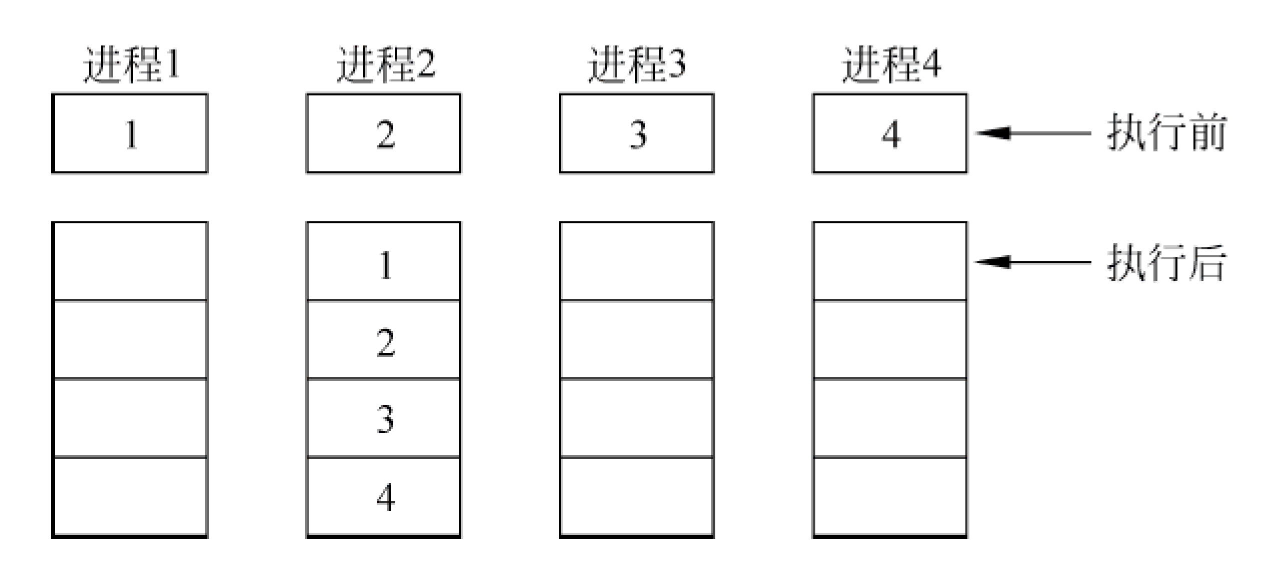


图 3-18　MPI_Gather 通信的示意图

2) MPI_Reduce

函数原型如下：

```
int MPI_Reduce(void * sendbuf, void * recvbuf, int count, PI_Datatype datatype, MPI_Op op,
               int root, MPI_Comm comm)
    IN          sendbuf             发送消息缓冲区的起始地址
    OUT         recvbuf             接收消息缓冲区中的地址
    IN          count               发送消息缓冲区中的数据个数
    IN          datatype            发送消息缓冲区的元素类型
    IN          op                  操作类型
    IN          root                根进程的标识号
    IN          comm                通信域
```

在 MPI_Reduce 操作中，每个进程(包括根进程本身)将其发送缓冲区中的消息发送

到给定的根(root)进程,根进程将接收到的各个进程发来的数据按给定的操作 op 进行运算,并将运算结果返回到根进程的输出缓冲区中。由于所有组成员都用同样的参数 count、datatype、op、root 和 comm 来调用该函数,因此所有进程都向根进程提供长度相同、元素类型相同的输入和输出缓冲区。每个进程可能向根进程提供一个或一系列元素以执行 op 操作。MPI_Reduce 的工作示意图如图 3-19 所示(图 3-19 中,执行的操作 op 为 MPI_SUM)。

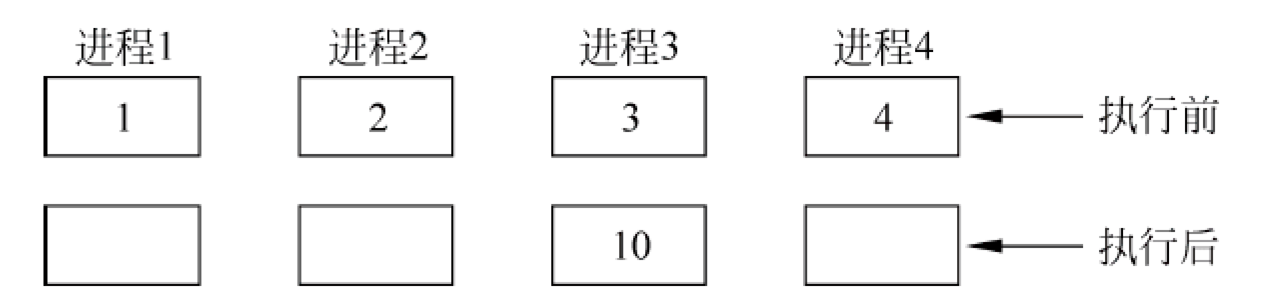


图 3-19 MPI_Reduce 通信的示意图

MPI 默认地定义了一些常用的规约操作,如表 3-6 所示。此外,根据自己的需要,用户还可以用 MPI_Op_create 函数创建新的规约操作。

表 3-6 MPI 预定义的常用规约操作

MPI 规约操作		C 语言数据类型
MPI_MAX	求最大值	integer, float
MPI_MIN	求最小值	integer, float
MPI_SUM	和	integer, float
MPI_PROD	乘积	integer, float
MPI_LAND	逻辑与	integer
MPI_BAND	按位与	integer, MPI_BYTE
MPI_LOR	逻辑或	integer
MPI_BOR	按位或	integer, MPI_BYTE
MPI_LXOR	逻辑异或	integer
MPI_BXOR	按位异或	integer, MPI_BYTE
MPI_MAXLOC	最大值和存储单元	float, double, long double
MPI_MINLOC	最小值和存储单元	float, double and long double

3) 程序实例

```
int rank,size,i,num[100];
MPI_Init(&argc,&argv);            /* 初始化 mpi,程序的开始并行执行 */
MPI_Comm_rank(comm,&rank);        /* 获得当前进程的标识号 */
MPI_Comm_size(comm,&size);        /* 获得总的进程数 */
/* 进程 0 从通信域中的所有进程收集数据并存储在数组 num 中 */
MPI_Gather(&rank,1,MPI_INT,num,1,MPI_INT,0,comm);
if(rank == 0)
{
```

```
    printf("Process 0 gather from other Process:\n");
    for(i = 0;i < size;i++)
    {
        printf(" % 4d",num[i]);
        if((i + 1) % 4 == 0)
            printf("\n");
    }
    printf("\n");
}
```

程序运行结果如下所示：

```
Process 0 gather from other Process:
   0   1   2   3
   4   5   6   7
```

3. 多对多集合通信函数

1）MPI_Allgather

函数原型如下：

```
int MPI_Allgather(void * sendbuf, int sendcount, MPI_Datatype sendtype, void * recvbuf, int
                  recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```

IN	sendbuf	发送消息缓冲区的起始地址
IN	sendcount	发送消息缓冲区中的数据个数
IN	sendtype	发送消息缓冲区中的数据类型
OUT	recvbuf	接收消息缓冲区的起始地址
IN	recvcount	从其他进程中接收的数据个数
IN	recvtype	接收消息缓冲区的数据类型
IN	comm	通信域

在 MPI_Allgather 操作中，每个进程都收集其他所有进程(包括该进程自己)发来的消息，相当于每个进程都执行了一次 MPI_Gather 操作。在执行完 MPI_Allgather 之后组内所有进程的接收缓冲区中的数据都是相同的。其工作示意图如图 3-20 所示。

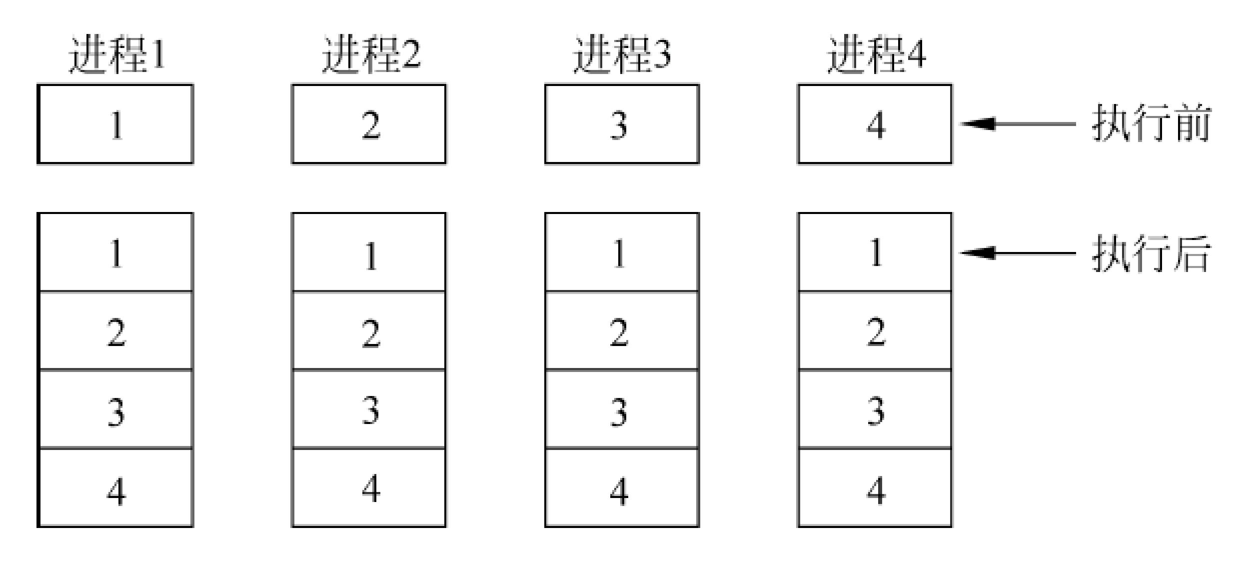


图 3-20　MPI_Allgather 通信的示意图

2）MPI_Allreduce

函数原型如下：

```
int MPI_Allreduce(void * sendbuf, void * recvbuf, int count, MPI_Datatype datatype, MPI_Op
                  op, MPI_Comm comm)
    IN          sendbuf         发送消息缓冲区的起始地址
    OUT         recvbuf         接收消息缓冲区的起始地址
    IN          count           发送消息缓冲区中的数据个数
    IN          datatype        发送消息缓冲区中的数据类型
    IN          op              操作类型
    IN          comm            通信域
```

MPI_Allreduce 操作相当于先执行 MPI_Reduce 操作，然后再执行 MPI_Bcast 操作。其工作示意图如图 3-21 所示（图 3-21 中，执行的操作 op 为 MPI_SUM）。

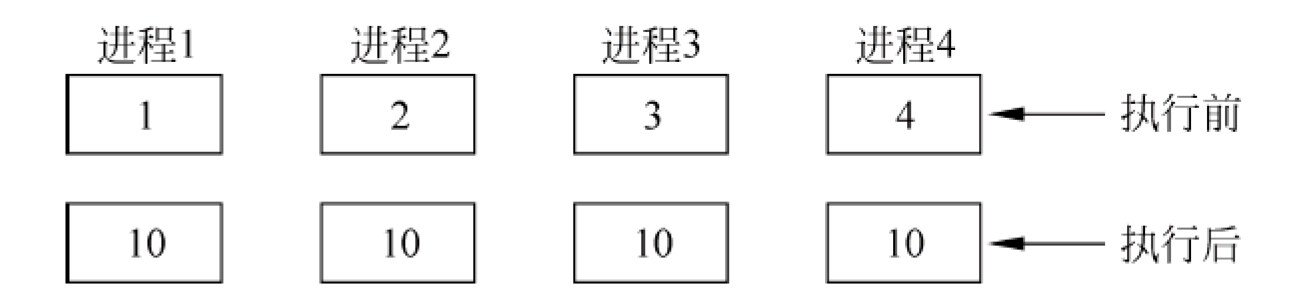


图 3-21　MPI_Allreduce 通信的示意图

3）MPI_Alltoall

函数原型如下：

```
int MPI_Alltoall(void * sendbuf, int sendcount, MPI_Datatype sendtype, void * recvbuf, int
                 recvcount, MPI_Datatype recvtype, MPI_Comm comm)
    IN          sendbuf         发送消息缓冲区的起始地址
    IN          sendcount       发送到每个进程的数据个数
    IN          sendtype        发送消息缓冲区中的数据类型
    OUT         recvbuf         接收消息缓冲区的起始地址
    IN          recvcount       从每个进程中接收的元素个数
    IN          recvtype        接收消息缓冲区的数据类型
    IN          comm            通信域
```

MPI_Alltoall 是在域内进程之间进行完全的消息交换。其中，每一个进程都向其他所有进程发送消息，同时，每一个进程都从其他所有进程接收消息。调用 MPI_Alltoall 相当于每个进程依次将它的发送缓冲区的第 i 块数据发送给第 i 个进程，同时每个进程又都依次从第 j 个进程接收数据并放到各自接收缓冲区的第 j 块数据区的位置。其工作示意图如图 3-22 所示。

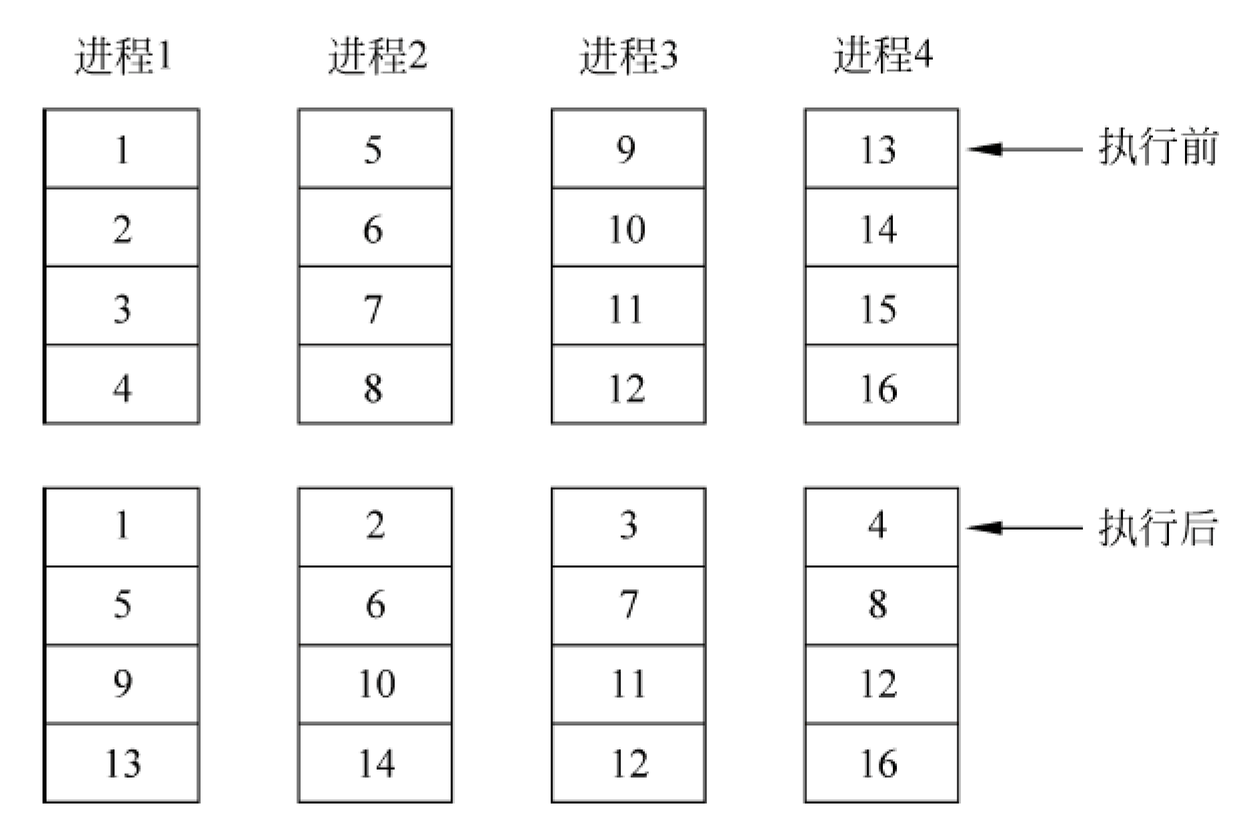


图 3-22 MPI_Alltoall 通信的示意图

4）MPI_Scan

函数原型如下：

```
int MPI_Scan(void * sendbuf, void * recvbuf, int count, MPI_Datatype datatype, MPI_Op op,
             MPI_Comm comm)
    IN          sendbuf           发送消息缓冲区的起始地址
    OUT         recvbuf           接收消息缓冲区的起始地址
    IN          count             输入缓冲区中元素的个数
    IN          datatype          输入缓冲区中元素的类型
    IN          op                操作类型
    IN          comm              通信域
```

可以将 MPI_Scan 看作一种特殊的归约，即每一个进程都对排在它前面的进程进行归约操作。MPI_Scan 调用的结果是，就进程 i 来说，它将进程 0、进程 1 一直到进程 i 的发送缓冲区的数据进行指定的归约操作，并将结果存入进程 i 的接收缓冲区。其工作示意图如图 3-23 所示（图 3-23 中，执行的操作 op 为 MPI_SUM）。

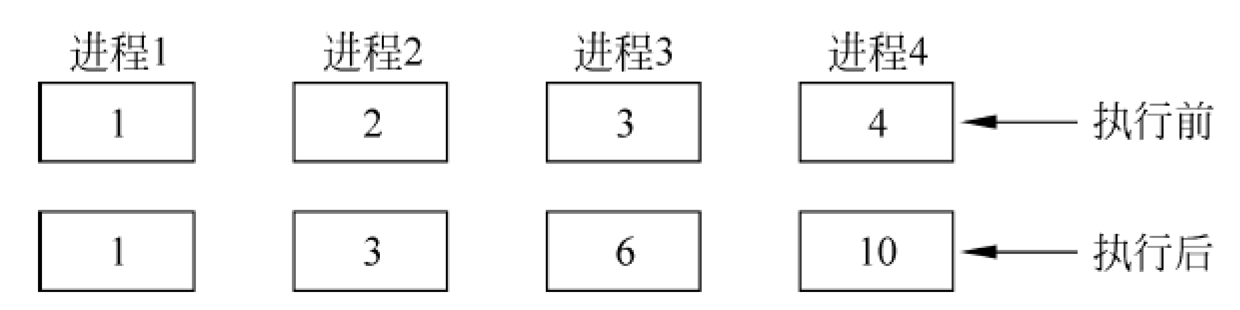


图 3-23 MPI_Scan 通信的示意图

5）程序示例

```
int rank,size,buf;
MPI_Init(&argc,&argv);          /* 初始化 mpi,并行程序开始执行 */
MPI_Comm_rank(comm,&rank);      /* 获得当前进程的标识号 */
MPI_Comm_size(comm,&size);      /* 获得总的进程数 */
```

```
/* 各进程将所有进程的 rank 值进行求和归约,然后存储在各自进程的 buf 中 */
MPI_Allreduce(&rank,&buf,1,MPI_INT,MPI_SUM,comm);
printf("Process %d have executed allreduce : buf = %d\n",rank,buf);
```

程序运行结果如下所示：

```
Process 0 have executed allreduce : buf=6
Process 2 have executed allreduce : buf=6
Process 1 have executed allreduce : buf=6
Process 3 have executed allreduce : buf=6
```

4. 同步操作 MPI_Barrier

MPI_Barrier 可在进程组中建立一个同步栅栏，当每个进程都到达该同步栅栏之后，程序才接着往下执行，只要有一个进程未到达该同步栅栏，则所有其他已到达该同步栅栏的进程都必须等待。同步的作用是当进程完成同步调用之后，可以保证所有的进程都已执行了同步点之前的操作。

函数原型如下：

```
int MPI_Barrier(MPI_Comm comm)
    IN          comm       通信域
```

MPI_Barrier 阻塞所有的调用者直到所有的进程组成员都调用了它，这之后，各个进程中的该调用才可以返回。

3.1.5　并行 I/O

当前，许多并行计算机能提供充分的硬件资源和高性能的并行文件系统，它们支持多个进程并行访问同一个文件。为此，MPI 提供了并行 I/O 函数，使得 MPI 程序中的多个进程能够并行地访问同一文件，并获得并行计算机提供的高性能文件系统服务。MPI 根据不同的文件访问特征，将并行 I/O 方法分为简单并行 I/O、显示偏移并行 I/O、非连续访问并行 I/O、聚合并行 I/O、阵列并行 I/O、非阻塞并行 I/O 和共享文件指针并行 I/O 等。下面，对部分常用的并行 I/O 方法进行简要介绍。

1. 简单的并行 I/O

1) 基本函数介绍

首先介绍一下并行 I/O 常用的基本函数，如表 3-7 所示。

表 3-7　并行 I/O 常用的基本函数

函数名	函数原型
打开文件	MPI_File_open(comm, filename, mode, info, myfile)
文件寻址	MPI_File_seek(fh, offset, whence)

续表

函数名	函数原型
文件读取	MPI_File_read(fh， * buf，count，datetype， * status)
文件写入	MPI_File_write(fh， * buf，count，datetype， * status)
关闭文件	MPI_File_close(* myfile)

各函数参数汇总详解如表 3-8 所示。

表 3-8 函数中的参数

参数名	参数解释
comm	进程所属的通信子对象
filename	进程打开或关闭的文件名
mode	文件的创建或访问方式
info	各进程交给 MPI 系统的附加提示信息
myfile	各进程获得的文件连接器
fh	文件连接器
offset	文件指针的偏移量
whence	文件指针偏移量的起始位置
buf	待写入或读取的数据缓冲区的起始位置
count	待写入或读取的数据单元的个数
datetype	待写入或读取的数据单元的类型
status	数据写入或读取的状态信息

函数 MPI_File_open 的第 3 个参数 mode 表示文件打开时，赋予该进程访问该文件的权限。MPI 支持的文件权限如表 3-9 所示。

表 3-9 MPI 支持的文件权限

Mode	文件权限
MPI_MODE_CREATE	如果文件不存在，则创建该文件
MPI_MODE_RDONLY	只读
MPI_MODE_WRONLY	只写
MPI_MODE_RDWR	读写
MPI_MODE_EXCL	如果要创建的文件已存在则报错
MPI_MODE_DELETE_ON_CLOSE	关闭时删除文件
MPI_MODE_UNIQUE_OPEN	不允许同时打开文件
MPI_MODE_SEQUENTIAL	顺序方式访问文件
MPI_MODE_APPEND	所有文件指针指向文件末尾

注意，文件权限可以叠加使用，这样的文件就有了多重权限，如 MPI_MODE_CREATE| MPI_MODE_RDWR。但是，MPI_MODE_CREATE 不可与 MPI _MODE_EXCL 合用，MPI _MODE_SEQUENTIAL 不可和 MPI_MODE_RDWR 合用。

至此，前面介绍的 5 个 MPI 并行 I/O 函数已经能够满足 MPI 并行应用程序的任何 I/O 需求。此外，MPI 还提供了一系列其他并行 I/O 函数，以简化并行编程、提高 I/O 性能和增强程序的可移植性，将在后续章节对此进行简要介绍。

2）程序示例

在下面程序中，启动两个进程，以共享文件指针的形式，分别向当前文件夹下的 file 文件写入"Hello"和"World"字串。该程序的代码如下。

```
MPI_File myfile;
int rank;
char buf[5];
MPI_Init(&argc,&argv);            /* 初始化 mpi,并行程序开始执行 */
MPI_Comm_rank(comm,&rank);        /* 获得当前进程的标识号 */
/* 各进程打开相同的文件,获得各自的文件连接器 */
int res = MPI_File_open(comm,"./file",MPI_MODE_RDWR,MPI_INFO_NULL,&myfile);
if(res)                           /* 如果打开文件失败 */
{
    printf("Unable open file \"./file\"\n");
}
else
{
    /* 确定文件连接器中文件指针在文件中的位置 */
    MPI_File_seek(myfile,rank * 5 * sizeof(char),MPI_SEEK_SET);
    if(rank == 0)
        buf = "Hello";
    else if(rank == 1)
        buf = "World";
    /* 各进程将缓冲区 buf 中的内容写入该文件中 */
    MPI_File_write(myfile,buf,5,MPI_CHAR,&status);
    printf("Process %d have written to the file: %s\n",rank,buf);
    MPI_File_close(&myfile);      /* 各进程释放各自的文件连接器 */
}
```

程序运行结果如下所示：

```
Process 0 have written to the file:Hello
Process 1 have written to the file:World
```

2. 显示偏移并行 I/O

上一节介绍了 MPI_File_read 和 MPI_File_write 函数具有相同的特征，即各个进程

拥有独立的文件指针,而且数据的读写必须从文件指针指定的初始位置开始。为了简化文件指针的定位,MPI 系统提供了另外两个 I/O 函数:MPI_File_read_at 和 MPI_File_write_at。它们能够直接从文件的任意给定位置开始读写数据,而且,不需要借助函数 MPI_File_seek 在读写操作执行之前为文件指针定位,这种操作称为显示偏移 I/O 函数。

1) 基本函数介绍

- 调用 MPI 显示偏移 I/O 读函数 MPI_File_read_a,各个进程可从指定位置开始读取文件中的数据。

函数原型如下:

```
int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void * buf, int count, MPI_Datatype
                     datatype, MPI_Status * status)
    IN          fh              文件连接器
    IN          offset          读取位置相对于文件头的偏移量
    OUT         buf             读取数据存放的缓冲区
    IN          count           读取数据个数
    IN          datatype        读取数据的数据类型
    OUT         status          返回的状态参数
```

- 调用 MPI 显示偏移 I/O 写函数 MPI_File_write_at,各个进程可从指定位置开始将数据写入文件中。

函数原型如下:

```
int MPI_File_write_at(MPI_File fh, MPI_Offset offset, void * buf, int count, MPI_Datatype
                      datatype, MPI_Status * status)
```

(它的参数定义与 MPI_File_read_at 相同)

函数 MPI_File_read_at 表示各个进程从 fh 连接的文件的第一个 offset 个字节开始,连续读取 count 个类型为 datatype 的数据单元,并将其存储在 buf 中。同样,MPI_File_write_at 函数表示各个进程从 fh 连接的文件的第 offset 个字节开始,将数组 buf 包含的连续 count 个类型为 datatype 的数据单元写入文件中。

2) 程序实例

在下面的程序中,启动两个进程,以独立文件指针显示偏移的形式,读取当前文件夹下的 file 文件,file 中的内容为"HelloWorld"。该程序的代码如下。

```
MPI_File myfile;
int rank;
char buf[5];
MPI_Init(&argc,&argv);          /* 初始化 mpi,程序开始并行执行 */
MPI_Comm_rank(comm,&rank);      /*获得当前进程的标识号 */
/* 各进程打开相同的文件,获得各自的文件连接器 */
```

```
int res = MPI_File_open(comm,"./file",MPI_MODE_RDWR,MPI_INFO_NULL,&myfile);
if(res)                         /* 如果打开文件失败 */
    printf("Unable open file \"./file\"\n");
else
{
        /* 以 buf 为缓冲区,各个进程从文件中读取 5 个 MPI_CHAR 类型的数据 */
        MPI_File_read_at(myfile,rank * 5 * sizeof(char),buf,5,MPI_CHAR,&status);
        printf("Process %d have readed from the file: %s\n",rank,buf);
        MPI_File_close(&myfile);    /* 各个进程释放各自的文件连接器 */

}
```

程序运行结果如下所示：

```
Process 0 have readed from the file:Hello
Process 1 have readed from the file:World
```

3. 其他文件操作方式

除了上述两种并行 I/O 操作之外，MPI 还提供了其他更高效的并行 I/O 操作方式，由于篇幅有限，下面只对其进行简要介绍。

1）非连续访问并行 I/O

前面介绍的两种并行 I/O 模式都默认对文件中连续的数据单元进行。但是，假设 MPI 程序中某些进程需要访问该文件中的 n 个数据单元，而这 n 个数据单元在文件中是非连续存储的。如果直接使用 MPI_File_read 函数或 MPI_File_read_at 函数逐个进行访问，那么一共需要 n 次 I/O 操作，显然这样做的效率比较低。为此，MPI 系统提供了文件窗口的概念。

给定一个文件，函数 MPI_File_set_view 可以确定该文件中哪些数据可以被某个进程访问，并称这些数据为该进程在该文件中获得的文件窗口。在文件窗口中，可以认为所有的数据单元是连续存储的。所以，如果某个进程要访问一个文件中的非连续存储的数据单元，可以给该进程设置一个文件窗口来包含所有要访问的数据单元。这样，该进程对这些数据单元的访问就等价于对该文件窗口中连续数据单元的访问。

调用 MPI_File_set_view 函数可定义文件窗口，函数原型如下：

```
int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype, MPI_Datatype
                      filetype, char * datarep, MPI_Info info)
    INOUT       fh          对应文件的文件连接器
    IN          disp        在文件中的偏移位置
    IN          etype       基本数据类型
    IN          filetype    文件类型
    IN          datarep     数据的表示方法
    IN          info        传递给运行时的信息
```

2）聚合并行 I/O

在同一时刻，如果有多个进程访问同一个文件，则可能导致多次小数据量的 I/O 访问，这将会影响并行 I/O 的性能。为此，MPI 系统提供了聚合 I/O 函数，它负责协调执行 I/O 访问的多个进程之间的关系，并将多个进程需要访问的数据聚合在一起，只进行一次文件访问操作，从而提高并行 I/O 的性能。

调用 MPI 聚合 I/O 读函数 MPI_File_write_all，各个进程从同一文件中读取数据。函数原型如下：

```
int MPI_File_read_all(MPI_File fh, void * buf, int count, MPI_Datatype datatype, MPI_Stat
                      us * status)
    IN          fh          文件连接器
    OUT         buf         读取数据存放的缓冲区
    IN          count       读取数据个数
    IN          datatype    读取数据的数据类型
    OUT         status      返回的状态参数
```

MP 聚合 I/O 写函数，各进程将各自数据写入同一文件中。函数原型如下：

```
int MPI_File_write_all(MPI_File fh, void * buf, int count, MPI_Datatype datatype, MPI_Stat
                       us * status)
    IN          fh          文件连接器
    OUT         buf         读取数据存放的缓冲区
    IN          count       读取数据个数
    IN          datatype    读取数据的数据类型
    OUT         status      返回的状态参数
```

3）共享文件指针

在前面所讲的所有并行 I/O 操作中，每个进程都拥有自己各自独立的文件指针，每个进程的文件指针的变化不会影响其他进程的文件指针。除此之外，MPI 系统还提供了共享文件指针的方法，即所有的进程对同一个文件进行操作时共享同一个文件指针。

MPI 系统提供了函数 MPI_File_read_shared 和 MPI_File_write_shared，负责各个进程从共享文件指针所指的当前位置开始读写文件中的数据。各个进程通过 MPI_File_seek_shar ed 函数显示地移动共享文件指针。这 3 个函数的参数列表分别与 MPI_File_read、MPI_File _wri te 和 MPI_File_seek 的参数列表完全相同。

4）非阻塞并行 I/O

前面所讲的所有并行 I/O 函数均采用阻塞方式，即只有当这些函数执行的 I/O 访问完成之后才能返回。除此之外，MPI 还提供了非阻塞式的并行 I/O 操作方法。具体函数见表 3-10。

表 3-10　非阻塞式的并行 I/O 函数

阻塞式并行 I/O 函数	非阻塞式并行 I/O 函数
MPI_File_read	MPI_File_iread
MPI_File_write	MPI_File_iwrite
MPI_File_read_at	MPI_File_iread_at
MPI_File_write_at	MPI_File_iwrite_at
MPI_File_read_shared	MPI_File_iread_shared
MPI_File_write_shared	MPI_File_iwrite_shared

3.1.6　MPI 应用实例

下面给出一个计算 pi 的程序，供大家参考。程序的代码如下。

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
double fun(double x);            /* 定义函数 fun(x) */
int main (int argc,char * argv[])
{
    MPI_Comm comm = MPI_COMM_WORLD;
    int n = 0,rank,size,i;
    double PI25DT = 3.141592653589793238462643;
    double mypi,pi,h,sum,x;
    MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    if (rank == 0)
    {
        printf("Please give N = ");
        scanf(" %d",&n);
    }
    /* 进程 0 广播 n 的值 */
    MPI_Bcast(&n,1,MPI_INT,0,comm);
    h = 1.0/(double) n;
    sum = 0.0;
    /* 每一个进程计算一部分矩形的面积，若总的进程数 size 为 4 的话，则将 0～1 区
     * 间划分为 100 个矩形. 各个进程分别计算矩形块
     * 进程 0: 1,5,9,13,…,97
     * 进程 1: 2,6,10,14,…,98
     * 进程 2: 3,7,11,15,…,99
     * 进程 3: 4,8,12,16,…,100 */
  for(i = rank + 1;i <= n;i += size)
```

```
    {
          x = h * ((double)i - 0.5);
          sum += fun(x);
    }
    mypi = h * sum;                /* 各进程并行计算得到的部分和 */
    /* 进程 0 执行归约操作,将各个进程计算出的部分和累加,得到所有矩形的面积,
     * 该面积和即为近似 PI 值 */
    MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,MPI_SUM,0,comm);
    if(rank == 0)
    {
        printf("pi is approximately %.16f\n",pi);
        printf("Error is %.16f\n",fabs(pi-PI25DT));
      }
  MPI_Finalize();
  }
  double fun(double x)
  {
      return(4.0/(1.0 + x * x));
  }
```

程序运行如下所示：

```
Please give N=1000
pi is approximately 3.1415927369231262
Error is 0.0000000833333331
```

本节小结

>>>

本节详细介绍了 MPI 的特点及其发展历程,然后以一个简单的 MPI 程序为例讲解了 MPI 编程环境的安装与配置,并介绍了一些常用的 MPI 基本函数。此外本节还针对 MPI 编程模型中的点对点通信、集合通信和并行 I/O 方法做了详细的讲解,并通过一些简单的程序实例向读者展示了相关函数的使用方法。

由于篇幅有限,本节只介绍了一些 MPI 的基本编程方法,其他诸如 MPI 进程组与通信域管理、进程拓扑、MPI 环境管理等更深层次技术在这里没有做进一步讲解。如对这部分内容感兴趣的话,读者可以参考其他 MPI 教材。

MPI 作为一种成熟的并行编程模型已在众多领域得到广泛的应用,希望读者通过本节内容的介绍,能对其有一定的了解,并能将其成功运用到实际的工程实践当中。

3.2 OpenMP

OpenMP(Open Multi-Processing)是适用于共享内存多处理器体系结构的可移植并行编程模型。其应用程序接口由 SGI 公司发起,由一些主要的计算机硬件与软件厂商制

定并得到认可。其规范由“OpenMP 体系结构审议委员会(Architecture Review Board, ARB)”创立并发布。

目前,OpenMP 规范的最新版本是 3.0,支持 Fortran、C 与 C++。支持 OpenMP 的编译器包括 Sun Studio、Intel Compiler、开放源码的 GCC 和 Open64 编译器等。OpenMP 能够支持多种平台,包括大多数的类 UNIX 系统(包括 Linux)以及 Windows NT 系统(包括 Windows 2000、Windows XP、Windows Vista 等)。

OpenMP 是通过编译指导、函数调用和环境变量的方式,显式地指导编译器怎样、什么时候利用应用程序的并行性,而无须程序员手工处理复杂的诸如线程创建、同步、负载均衡和销毁等技术细节。

3.2.1　OpenMP 简介

1. OpenMP 的发展历程

1994 年,提出第一个 ANSI X3H5 草案,被否决;

1997 年,OpenMP 标准规范代替原先被否决的 ANSI X3H5,被认可;

1997 年 10 月,发布与 Fortran 语言捆绑的第一个标准规范 FORTRAN version1.0;

1998 年 11 月 9 日,发布支持 C 和 C++的标准规范 C/C++ version1.0;

2000 年 11 月,发布 FORTRAN version2.0;

2002 年 3 月,发布 C/C++ version2.0;

2005 年 5 月,发布 OpenMP2.5,将原来的 Fortran 和 C/C++标准规范相结合;

2008 年 3 月,发布 OpenMP3.0 标准。

相关规范可在 http://www.openmp.org/drupal/node/view/8 中下载。

2. 为什么使用 OpenMP

OpenMP 的应用程序接口(API)是在共享存储体系结构上的一个编程模型,它包含编译指导(Compiler Directive)、运行函数库(Runtime Library)和环境变量(Environment Variables)三部分。

OpenMP 是编译指令与库函数的集合,这些编译指令和库函数主要用于创建共享存储计算机的并行程序。OpenMP 组合了 C、C++或 Fortran,以创建一种多线程编程语言。它的语言模型基于这样一种假设:假设执行单元是共享同一地址空间的线程。

OpenMP 具有两个特性:串行等价性和增量并行性。

- 串行等价性:当一个程序无论是使用一个线程运行还是使用多个线程运行时,它都能够产生相同的结果,则该程序具有串行等价性。在大多数情形下,具有串行等价性的程序更易于维护和理解(因此也更容易编写)。

• 增量并行性：这是一种并行的编程类型。处理器从一个串行程序开始，一块接着一块地寻找那些值得并行化的代码段。这样，并行性被逐渐地添加。在该过程的每个阶段，存在一个可以被验证的程序，这极大地增加了程序并行化的成功概率。

3. OpenMP 的 Fork-Join 模型

OpenMP 的执行模型采用 Fork-Join 的形式，以线程为基础，通过编译指导语句显式地指导并行化，为编程人员提供了对并行化过程的完全控制，如图 3-24 所示。

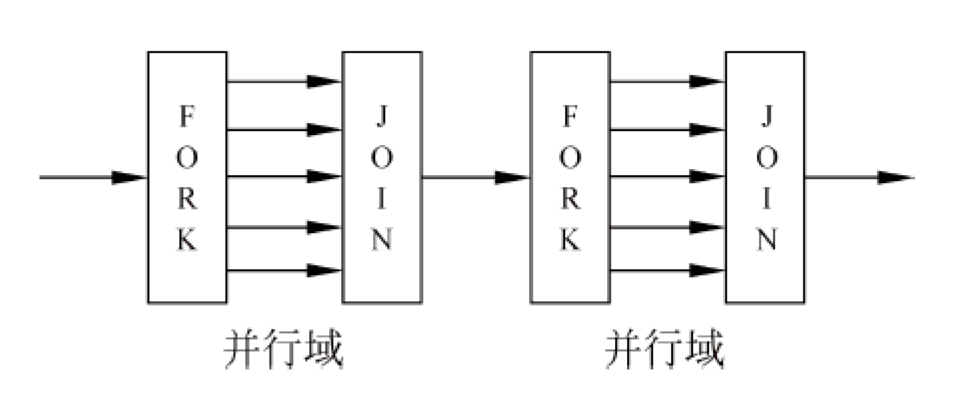


图 3-24 Fork-Join 模型

线程是具有独立执行命令能力的运行单位，当操作系统创建一个进程来执行程序时，它会给这个进程分配一定的资源，比如内存空间和寄存器资源。当一个进程的多个线程共同执行一个程序时，则各个线程会共享它们所属的进程的资源，比如内存地址空间。单独的线程只需要很少的资源，例如程序计数器和一块存储其私有变量(如寄存器和堆栈)的内存区域。

多个线程可以通过上下文切换的方式在单处理器或者单核心上执行，或者以并发多线程的方式交替执行。当多个线程同时在多处理器或者多核心上工作时，它们可以同时(并行)地执行并行程序。

编写多线程程序的方法有很多，其中一些方法支持复杂的线程通信。OpenMP 提供了一种结构化的多线程编程方法，该方法更简便，同时可以最大程度地帮助程序员避免潜在的错误。在这种编程方法中，首先就像串行程序那样，以单线程方式执行程序，该线程被默认为初始线程。当初始线程遇到 OpenMP 的并行结构语句时，会创建一个线程组(就是 fork 动作)，而自己作为该线程组的主线程(Master)。主线程与线程组中的从线程(Slaver)根据程序的结构动态地分配任务。在退出并行结构时，只有原始线程(即主线程)继续执行，其他所有的从线程结束执行(就是 join 动作)。

使用 OpenMP 进行并行编程，程序开发者能够设计出高质量的并行代码，同时有机会对并行算法进行创新。它提供了用来标记 OpenMP 程序并行区域的语句(代码中用并行结构标记的部分称为并行区域)。同时，它还提供了控制这些并行语句如何并行执行的附加信息。

OpenMP 降低了并行编程的难度与复杂度，程序员可以把更多的精力投入到并行算法本身，而非其具体的实现细节。对基于数据分集(fork-join)的多线程程序设计来说，OpenMP 是一个很好的选择。另外，OpenMP 还提供了更强的灵活性，易于适应不同的并行系统配置。并行粒度、负载均衡等是传统多线程程序设计中需要解决的难题，OpenMP 库从程序员手中接管了上述部分工作。

读者可从 OpenMP 的官方网站 http://www.openmp.org 上查询它的最新发展和相关资料。

3.2.2　第一个 OpenMP 程序

1. Windows 下支持 OpenMP 的编译器

Microsoft Visual Studio 2008 完全支持 OpenMP 2.0 标准，无须额外安装其他软件。通过新的编译选项/openmp 支持 OpenMP 程序的编译，编译器会自动将用户编写的 OpenMP 程序与 Windows 下实现的 OpenMP 库 vcomp.dll 连接在一起。OpenMP 程序在运行时，会自动寻找 vcomp.dll。下面，使用 Visual Studio 2008 来新建一个 OpenMP 项目 OpenMP。

启动 Visual Studio 2008，新建一个 Win32 控制台应用程序，并命名为 OpenMP，如图 3-25 所示。



图 3-25　新建项目

单击“确定”按钮，在详细设置向导里保持默认的设置即可，然后单击“完成”按钮。从而创建了一个控制台项目，如图 3-26 所示。

用鼠标右键单击项目，在弹出的菜单中选择“属性”。在弹出的对话框中找到“配置”→C/C++→“语言”→“OpenMP 支持”→“是(/OpenMP)”，单击“确定”按钮，实现对项目的 OpenMP 支持，如图 3-27 所示。



图 3-26 创建控制台项目

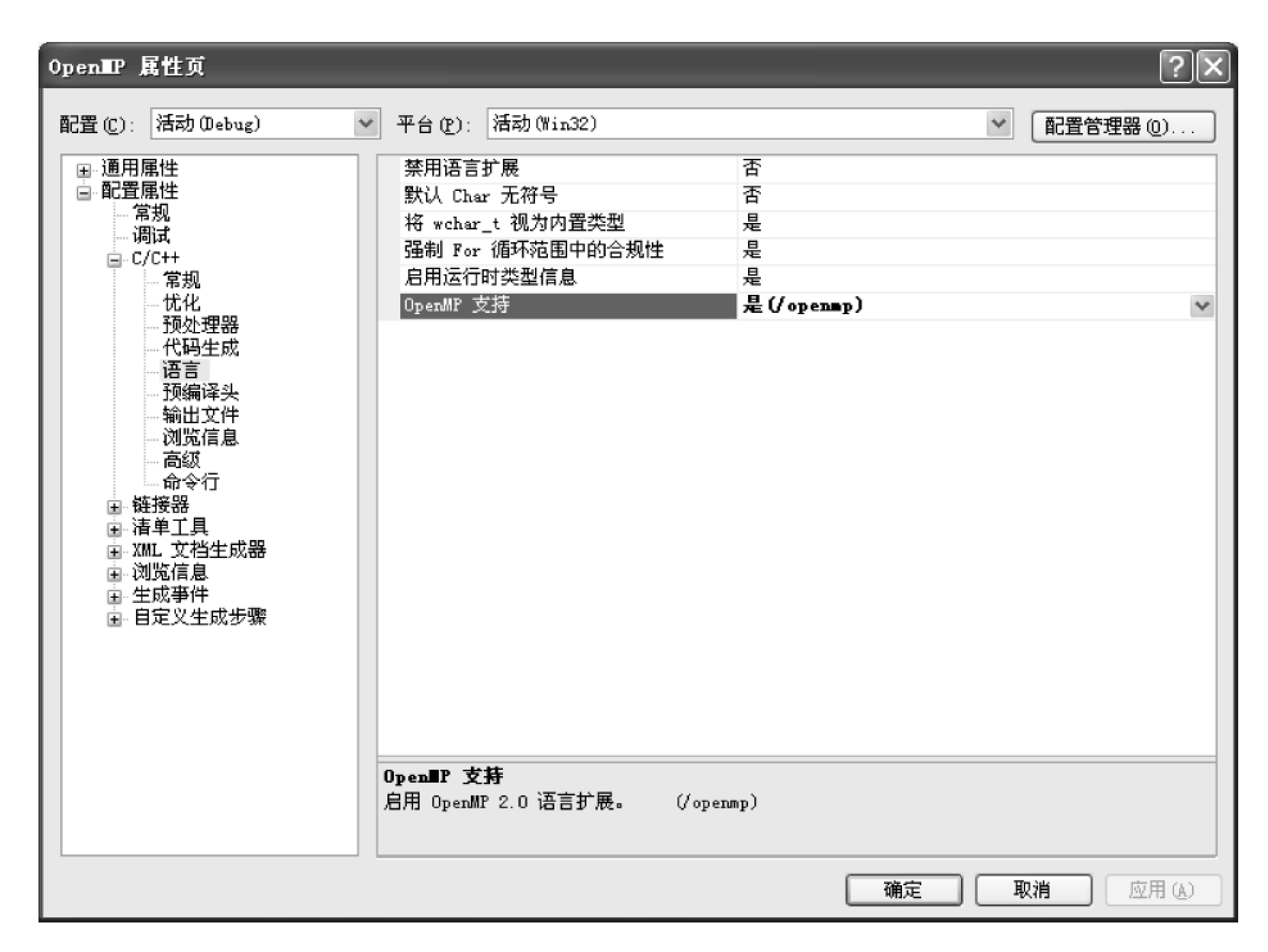


图 3-27 设置 OpenMP 支持

以上是 OpenMP 环境配置的一般方式。在实际的系统环境配置和编程软件安装过程中，可能存在其他运行中的障碍，请读者细心查阅其他相关资料。

2. Linux 下支持 OpenMP 的编译器

下面有关 GCC 简介与 GCC 的基本用法参考自文献[6]-[7]，有关使用 GCC 编译 OpenMP 程序参考自文献[8]。

1）GCC 简介

GCC(GNU Compiler Collection，GNU 编译器套装)，是一套由 GNU 开发的编程语言编译器。它是以 GPL、LGPL 许可证所发行的自由软件，也是 GNU 计划的关键部分，亦是自由的类 UNIX 及苹果计算机 Mac OS X 操作系统的标准编译器。

GCC 原名为 GNU C 语言编译器，这是因为它原来只能处理 C 语言。随后，GCC 很快得到扩展，可以处理 C++语言。之后，发展到可以处理 Fortran、Pascal、Objective-C、Java、Ada 及其他语言。

目前，GCC 编译器的最新稳定版本是于 2010 年 12 月 16 日发布的 4.5.2 版本。

2）GCC 的基本用法

在使用 GCC 编译器的时候，用户必须给出一系列必要的调用参数和文件名称。GCC 编译器的编译选项大约有 100 多个，其中的多数参数平时根本不会被使用。这里只介绍其中最基本、最常用的参数。

GCC 遵循的部分约定规则如下：

- 以.c 为后缀的文件是用 C 语言编写的源代码文件；
- 以.a 为后缀的文件是由目标文件构成的档案库文件；
- 以.C、.cc 或.cxx 为后缀的文件是用 C++语言编写的源代码文件；
- 以.h 为后缀的文件是程序中所包含的头文件；
- 以.i 为后缀的文件是已经预处理过的 C 源代码文件；
- 以.ii 为后缀的文件是已经预处理过的 C++源代码文件；
- 以.m 为后缀的文件是 Objective-C 源代码文件；
- 以.o 为后缀的文件是编译后的目标文件；
- 以.s 为后缀的文件是汇编语言源代码文件；
- 以.S 为后缀的文件是经过预编译的汇编语言源代码文件。

GCC 编译的基本格式如下：

```
gcc [options] [filenames]
```

其中，options 就是编译器的编译选项，filenames 给出相关的文件名称。

-c，只编译而不连接成为可执行文件，编译器只是将输入的.c 等源代码文件生成以

.o 为后缀的目标文件,通常用于编译不包含主程序的子程序文件。

-o output_filename,确定输出文件的名称为 output_filename。同时,该名称不能和源文件同名。如果不给出这个选项,gcc 就给出预设的可执行文件 a.out。

-g,生成调试工具(GNU 的 gdb)必要的符号信息,要想对源代码进行调试,就必须使用该选项。

-O,对程序进行优化编译与连接。使用该选项,会在编译与连接的过程中对整个源代码进行优化处理,这样生成的可执行文件的执行效率可以提高。不过,编译与连接的速度会相应慢一些。

-O2,比-O 更强的优化编译与连接。当然,整个编译与连接的过程会更慢。

-Idirname,将 dirname 指定的目录加入程序的头文件目录列表中,这是一个在预编译过程中使用的选项。C 程序中的头文件包含如下两种情况:

```
#include <myinc.h>
```

或

```
#include "myinc.h"
```

其中,第一种情况使用尖括号< >,第二种情况使用双引号" "。前者,预处理程序将在系统预设的包含文件目录(如/usr/include)中寻找相应的头文件;后者,预处理程序会在目标文件的文件夹内搜索相应的头文件。

3) 用 GCC 编译 OpenMP 程序

目前,能稳定支持 OpenMP 编译的 GCC 版本是 4.2.4 版。当然,新的版本也能很好地支持 OpenMP。使用 GCC 编译器来编译带有 OpenMP 编译指导语句的 C/C++程序,只需加上-fopenmp 选项即可,编译器会依照 OpenMP API v2.5 的规范来编译 OpenMP 程序。当使用-fopenmp 选项时,会默认包含-pthread 选项。因此,想使用-fopenmp 选项进行编译的 OpenMP 程序,必须同时支持使用-pthread 选项进行编译。

3. 第一个 OpenMP 程序

下面,以 Hello World 为例,给出 OpenMP 程序的一个简单框架,以便读者对 OpenMP 程序有一个初步的认识。

```
#include"omp.h"                          /* 包含头文件 omp.h */
#include<stdio.h>
#define NUM_THREADS 4

int main(int argc, char *argv[])
{
        int nthreads, tid;
```

```
        omp_set_num_threads(NUM_THREADS);       /* 为后面的并行区设置线程数目 */
        #pragma omp parallel                    /* 编译指导语句,创建一个并行区 */
        {
                tid = omp_get_thread_num();     /* 得到当前线程的标识号(线程号) */
                printf ("Hello World from OMP thread %d\n", tid);
                if(tid == 0)
                {
                        nthreads = omp_get_num_threads();
                        /* 得到并行区中总的线程数目 */
                        printf("Number of threads %d\n", nthreads);
                }
        }
        return 0;
}
```

下面,分析一下该 OpenMP 程序。

① 首先,C 语言的 OpenMP 程序需要包含 OpenMP 实现的头文件“omp. h”。

② 定义相关变量,nthreads 表示线程数,tid 表示当前线程的标识号。

③ 调用 OpenMP 的运行时库函数 omp_set_num_threads(),为后面的并行区设置线程数目。

④ 使用 OpenMP 的编译指导语句#pragma omp parallel,创建一个并行区。在该并行区中,多个线程共同执行。

⑤ 调用 OpenMP 的运行时库函数 omp_get_thread_num(),得到当前线程的标识号(即线程号)。

⑥ 调用 OpenMP 的运行时库函数 omp_get_num_threads(),得到该并行区中总的线程数目。

⑦ 最后由 0 号线程打印该并行区中总的线程数目。

下面是在 Linux 环境下,编译、连接并运行这个简单的 OpenMP 程序。

- 在当前文件夹下建立一个 test. c 文件,并编写上述代码。
- 使用 gcc 编译器、-fopenmp 编译选项编译并连接该文件。gcc -fopenmp -o test test. c,生成可执行程序 test。
- 运行可执行文件. /test。

运行结果如下所示:

```
Hello World from OMP thread 0
Number of threads 4
Hello World from OMP thread 2
Hello World from OMP thread 1
Hello World from OMP thread 3
```

3.2.3　编译指导语句

顾名思义,编译指导语句就是在编译过程中能够被编译器识别的特定注释语句。这

些特定的注释语句代表了 OpenMP 的语义。如果用户所使用的编译器不能识别 OpenMP 的编译语句,则这些特定的注释语句将会被当成普通的注释而被忽略。因此,如果仅仅使用编译指导语句,则编写的 OpenMP 程序既支持普通编译器,又支持 OpenMP 的编译器。这样做的好处就是,OpenMP 程序既可被看作并行程序又可被看作串行程序,或者在保持串行程序部分不变的情况下,用户能够方便地将串行程序改写成并行程序,在很大程度上方便了编程人员。

下面是 OpenMP 并行指导语句的基本格式:

```
#pragma omp <编译指导关键词> [子句[ [, ]子句] ...] 换行符
```

- #pragma omp 是编译指导关键词的前缀。所有的 OpenMP 并行指导语句都需要这样的前缀,属于固定部分。
- <编译指导关键词>是 OpenMP 的编译指导关键词。在前缀和子句之间必须有一个正确的编译指导关键词来决定当前编译指导语句的作用,属于必备的可变部分。OpenMP 合法的编译指导关键词包括 parallel、for、sections、single、parallel for、parallel sections、task、master、critical、barrier、taskwait、atomic、flush、ordered、threadprivate 等。
- [子句]主要负责并行执行部分中变量的共享与复制等。在没有其他约束的条件下,多个子句可以无序排列,也可以任意选择。由于编译器有自己的默认属性,所以该部分也可以没有,属于可选的部分。OpenMP 合法的子句有 default、shared、private、firstprivate、lastprivate、reduction、copyin、copyprivate。
- 换行符表明这条指导语句的终止。

1. 并行域结构

parallel 语句

parallel 语句会创建一个线程组来并行执行程序。其格式为:

```
#pragma omp parallel [子句[ [, ]子句] ...] 换行符
结构化块
```

支持的子句包括:

```
if(标量表达式)
num_threads(整型表达式)
default(共享 | 空)
private(变量列表)
firstprivate(变量列表)
shared(变量列表)
copyin(变量列表)
```

```
reduction(归约操作类型: 变量列表)
```

注意，当一个线程运行到 parallel 指令时，它会创建一个线程组并成为该组的主线程(Master)。主线程也是该组线程中的一员，其线程号为 0。当并行区开始时，程序代码将被复制，创建的线程组中的每个线程都会执行。

2. 共享任务结构

共享任务结构将它所包含的任务分配给线程组中各个线程执行。一个共享任务结构必须动态地封装在一个并行区中，以便该指导语句可以被并行执行。共享任务结构必须出现在一个线程组中或者根本不在该线程组中出现。连续的共享任务结构必须在线程组的所有线程中按相同次序出现。

1) for 循环结构语句

for 循环结构语句将循环语句中的重复部分分配给各个线程并行执行。其格式为：

```
#pragma omp for [子句[[,]子句] ... ] 换行符
结构化块
```

支持的子句：

```
private(变量列表)
firstprivate(变量列表)
lastprivate(变量列表)
reduction(operator: 变量列表)
schedule(kind[, chunk_size])
collapse(n)
ordered
nowait
```

在这里，暂不考虑各子句的含义，其具体的阐述参考下一小节。下面看一个有关 for 编译指导语句的实例，了解一下 for 编译指导语句的具体用法。

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main()
{
   int i, n=9;
   omp_set_num_threads(4);
#pragma omp parallel default(none) shared(n) private(i)    /* 创建并行区 */
{
   #pragma omp for    /* for 编译指导语句，每个线程分别执行下面 for 循环部分 */
```

```
    for (i = 0; i < n; i++)
        printf("Thread %d executes loop iteration %d\n",
                omp_get_thread_num(), i);
}   /* 并行区结束 */

    return(0);
}
```

运行结果如下所示：

```
Thread 0 executes loop iteration 0
Thread 0 executes loop iteration 1
Thread 0 executes loop iteration 2
Thread 2 executes loop iteration 6
Thread 2 executes loop iteration 7
Thread 2 executes loop iteration 8
Thread 1 executes loop iteration 3
Thread 1 executes loop iteration 4
Thread 1 executes loop iteration 5
```

2）sections 编译指导语句

sections 编译指导语句是非迭代的共享任务结构，它将其包含的任务分配给线程组中的各个线程。嵌套在 sections 编译指导语句中的不同的 section 指导语句，由线程组中不同的线程执行。sections 语句的格式为：

```
#pragma omp sections [子句[[,]子句] …] 换行符
{
    [#pragma omp section 换行]
    结构化块
    [#pragma omp section 换行]
    结构化块
}
```

支持的子句：

```
private(变量列表)
firstprivate(变量列表)
lastprivate(变量列表)
reduction(operator: 变量列表)
nowait
```

section 指导语句的用法如下：

```
#pragma omp parallel
{
   #pragma omp sections
   {
      #pragma omp section
        (void) funcA();
```

```
        #pragma omp section
          (void) funcB();
    }     /* sections 块结束 */

}      /* 并行区结束 */
```

3）single 编译指导语句

single 编译指导语句将其包含的代码交由线程组中的一个线程执行。对于必须各线程分别执行的代码(同时执行会造成计算结果错误)，该指导语句非常有用。具体的格式如下：

```
#pragma omp single [子句[[,] 子句] ...] 换行符
结构化模块
```

支持的子句：

```
private(变量列表)
firstprivate(变量列表)
copyprivate(变量列表)
nowait
```

除非使用了 nowait 子句，否则线程组中没有执行 single 语句的线程，将一直等到 single 指导语句包含的代码块执行结束。single 编译指导语句的用法如下：

```
#pragma omp parallel shared(a, b) private(i)
{
    #pragma omp single        /* 只有最先到达的那一个线程执行其包含的代码 */
    {
        a = 10;
        printf("Single construct executed by thread %d\n",
                omp_get_thread_num());
    }
    #pragma omp for
    for (i = 0; i < n; i++)
        b[i] = a;
}
    printf("After the parallel region:\n");
    for (i = 0; i < n; i++)
        printf("b[%d] = %d\n", i, b[i]);
```

3. 同步结构

在操作系统的相关课程中，学习过同步、临界区使用等知识。在多线程编程过程中，也会碰到线程同步、临界区使用等情况。

假设两个线程分别位于共享存储的不同处理器上，它们需要分别对共享存储中的某个变量 x 进行加 1 操作，该过程可能的执行顺序为：

- 线程 0 将变量 x 的值存入寄存器 A；
- 线程 1 将变量 x 的值存入寄存器 A；
- 线程 0 将寄存器 A 中变量 x 的值加 1；
- 线程 1 将寄存器 A 中变量 x 的值加 1；
- 线程 0 将寄存器 A 的值写回到变量 x 中；
- 线程 1 将寄存器 A 的值写回到变量 x 中。

最终，变量 x 的值为 1 而不是预期的 2。为了避免这类错误的产生，对变量 x 的修改操作必须在两个进程之间进行同步。

OpenMP 提供了多种同步结构负责执行过程中各个线程的同步。

1）barrier 编译指导语句

barrier 编译指导语句实现一个线程组中所有线程的同步。先到达 barrier 指导语句的线程将会被阻塞，直到所有其他线程都执行到该指导语句。其具体格式如下：

```
#pragma omp barrier 换行符
```

下面给出一个有关 barrier 编译指导语句的实例，了解一下其具体用法。

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <omp.h>

void print_time(int TID, char * comment);
int main()
{   int TID;
    int i, n = 10;
    int a[n], b[n], ref[n];
    (void) omp_set_num_threads(4);
    #pragma omp parallel private(TID)
    {
        TID = omp_get_thread_num();
        if ( TID < omp_get_num_threads()/2 )
            system("sleep 3");                  /* 当前进程睡眠 3 秒钟 */
        (void) print_time(TID, "before");       /* 调用 print_time 函数 */

        #pragma omp barrier
```

```
        (void) print_time(TID, "after ");
    }

    for (i = 0; i < n; i++)
    {
        b[i] = 2 * (i + 1);
        ref[i] = i + b[i];
    }

    #pragma omp parallel private(i) shared(n, a, b)
    {
        #pragma omp for schedule(dynamic, 1) nowait
        for (i = 0; i < n; i++)
            a[i] = i;

        #pragma omp barrier                    /* 对线程组中的所有线程进行同步 */

        #pragma omp for schedule(dynamic, 1) nowait
        for (i = 0; i < n; i++)
            a[i] += b[i];
    }

    printf("After the parallel region\n");
    for (i = 0; i < n; i++)
        printf(" a[%3d] = %6d ref[%3d] = %6d\n", i, a[i], i, ref[i]);
    return(0);
}

void print_time(int TID, char *comment)
{
    time_t tp;
    char buffer[26], mytime[9];
    (void) time(&tp);
    strcpy(&buffer[0], ctime(&tp));
    strncpy(&mytime[0], &buffer[11], 8);
    mytime[8] = '\0';
    printf("Thread %d %s barrier at %s\n", TID, comment, &mytime[0]);
    return;
}
```

运行结果如下所示：

```
Thread 3 before barrier at 13:08:43
Thread 2 before barrier at 13:08:43
Thread 0 before barrier at 13:08:46
Thread 1 before barrier at 13:08:46
Thread 1 after  barrier at 13:08:46
Thread 3 after  barrier at 13:08:46
Thread 2 after  barrier at 13:08:46
Thread 0 after  barrier at 13:08:46
After the parallel region
 a[  0] =      2 ref[  0] =      2
 a[  1] =      5 ref[  1] =      5
 a[  2] =      8 ref[  2] =      8
 a[  3] =     11 ref[  3] =     11
 a[  4] =     14 ref[  4] =     14
 a[  5] =     17 ref[  5] =     17
 a[  6] =     20 ref[  6] =     20
 a[  7] =     23 ref[  7] =     23
 a[  8] =     26 ref[  8] =     26
 a[  9] =     29 ref[  9] =     29
```

2）ordered 编译指导语句

ordered 编译指导语句包含在循环内，将被按序串行执行。其格式如下：

```
#pragma omp ordered 换行符
```

下面给出一个使用 order 编译指导语句的实例：

```
int i, TID, n = 9;
int a[n];
omp_set_num_threads(4);
for (i = 0; i < n; i++)
    a[i] = i;
#pragma omp parallel for default(none) ordered schedule(runtime) \
        private(i, TID) shared(n, a)
  for (i = 0; i < n; i++)
  {
      TID = omp_get_thread_num();
      printf("Thread %d updates a[%d]\n", TID, i);
      a[i] += i;
#pragma omp ordered                          /* 下面的打印语句将被按序串行执行 */
    {printf("Thread %d prints value of a[%d] = %d\n", TID, i, a[i]);}
  }
```

3）critical 编译指导语句

critical 编译指导语句指定代码段在同一时刻只能由一个线程执行。其具体的格式如下：

```
#pragma omp critical 换行符
```

当 critical 包含的代码段正被某个线程执行时，如果另一个线程也执行了 critical 指导语句，则它将被阻塞直到临界区中的线程退出。

下面给出一个使用 critical 编译指导语句的实例：

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define SUM_INIT 0

int main()
{
   int i, n = 25;
   int sum, TID, a[n];
   int ref = SUM_INIT + (n - 1) * n/2;
   int sumLocal;

   (void) omp_set_num_threads(3);

   for (i = 0; i < n; i++)
       a[i] = i;

   #pragma omp parallel
   {
       #pragma omp single
         printf("Number of threads is %d\n", omp_get_num_threads());
   }

   sum = SUM_INIT;
   printf("Value of sum prior to parallel region: %d\n", sum);
   #pragma omp parallel default(none) shared(n, a, sum) \
           private(TID, sumLocal)
   {
      TID = omp_get_thread_num();
      sumLocal = 0;
      #pragma omp for
        for (i = 0; i < n; i++)
          sumLocal += a[i];
      #pragma omp critical (update_sum)
      /* critical 编译指导语句包含的代码段在同一时刻只能由一个线程执行 */
      {
        sum += sumLocal;
        printf("TID =%d: sumLocal =%d sum =%d\n", TID, sumLocal, sum);
      }
   }

   printf("Value of sum after parallel region: %d\n", sum);
   printf("Check results: sum =%d (should be %d)\n", sum, ref);
```

```
    return(0);

}
```

运行结果如下所示：

```
Number of threads is 3
Value of sum prior to parallel region: 0
TID=0: sumLocal = 36 sum = 36
TID=1: sumLocal = 117 sum = 153
TID=2: sumLocal = 147 sum = 300
Value of sum after parallel region: 300
Check results: sum = 300 (should be 300)
```

4）atomic 编译指导语句

atomic 编译指导语句指定特定的存储单元将被原子地更新，不允许多个线程同时执行更新操作。其格式如下：

```
#pragma omp atomic 换行符
```

具体用法如下：

```
int ic, i, n = 7;
ic = 0;
#pragma omp parallel for shared(ic, n) private(i)
for (i = 0; i < n; i++)
{
   #pragma omp atomic                          /* ic 的修改是原子性的 */
     ic += 1;
}
printf("Counter = %d\n", ic);
```

5）master 编译指导语句

master 编译指导语句指定代码段将由主线程执行，该线程组中的其他线程将忽略该代码段。其格式如下：

```
#pragma omp master 换行符
```

其具体用法如下：

```
#pragma omp parallel shared(a, b) private(i)
{
  #pragma omp master                          /* 只能由 master 线程执行所包含的代码 */
  {
     a = 10;
     printf("Master construct is executed by thread %d\n",
            omp_get_thread_num());
  }
```

```
    #pragma omp barrier

    #pragma omp for
    for (i = 0; i < n; i++)
        b[i] = a;
}

    printf("After the parallel region:\n");
    for (i = 0; i < n; i++)
        printf("b[%d] =%d\n", i, b[i]);
```

6）threadprivate 编译指导语句

threadprivate 编译指导语句，将一个全局文件作用域变量的属性变为在并行区为每个线程所私有，其格式如下：

```
#pragma omp threadprivate (list) 换行符
```

其具体用法如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define TRUE  1
#define FALSE 0
int calculate_sum(int length);
int *pglobal;
#pragma omp threadprivate(pglobal)
/* 将全局文件作用域变量 pglobal 的属性变为在并行区内为每个线程所私有 */
int main()
{
   int i, j, sum, TID, n = 5;
   int length[n], check[n];
   omp_set_num_threads(3);
   for (i = 0; i < n; i++)
   {
      length[i] = 10 * (i + 1);
      check[i] = length[i] * (length[i] + 1)/2;
   }
   #pragma omp parallel for shared(n, length, check) private(TID, i, j, sum)
   for (i = 0; i < n; i++)
   {
      TID = omp_get_thread_num();
      if ( (pglobal = (int *) malloc(length[i] * sizeof(int))) != NULL ) {
         for (j = sum = 0; j < length[i]; j++)
              pglobal[j] = j + 1;
```

```
            sum = calculate_sum(length[i]);
            printf("TID %d: value of sum for i = %d is %8d (check = %8d)\n",
                   TID, i, sum, check[i]);

            free(pglobal);

        } else {
            printf("TID %d: fatal error in malloc for length[ %d] = %d\n",
                  TID, i, length[i]);
        }
    }
    return(0);
}
int calculate_sum(int length)
{
    int sum = 0;
    int j;
    for (j = 0; j < length; j++)
        sum += pglobal[j];
    return(sum);
}
```

运行结果如下所示：

```
TID 0: value of sum for i = 0 is       55 (check =       55)
TID 0: value of sum for i = 1 is      210 (check =      210)
TID 2: value of sum for i = 4 is     1275 (check =     1275)
TID 1: value of sum for i = 2 is      465 (check =      465)
TID 1: value of sum for i = 3 is      820 (check =      820)
```

3.2.4 数据共享属性子句

除了编译指导语句以外，另一个重点是数据属性。要学好 OpenMP 编程，就必须理解和会使用数据属性。由于 OpenMP 建立在共享存储编程模型之上，所以绝大部分变量默认为共享。

全局变量包括文件作用域变量和静态变量，私有变量包括循环计数(索引)变量和并行区调用的子程序中的堆栈变量。

OpenMP 的数据属性子句包括 private、firstprivate、lastprivate、shared、default、reduction 和 copyin 等。它与编译指导语句 parallel for 和 section 相结合，用来控制变量的作用范围。

1. default 子句

default 子句让用户自行定义在一个并行区的静态范围内变量的默认作用范围。其

格式如下：

```
#default(shared | private | none)
```

2. shared 子句

shared 子句表示它所列出的变量被线程组中所有线程共享。其具体格式如下：

```
shared(list)
```

共享变量在存储器中只有一份拷贝，所有的线程都能对它进行读写访问。对该变量访问的正确性将由程序员来保证。具体用法如下：

```
#pragma omp parallel for shared(a)      /* 变量 a 被该并行区中的所有线程共享 */
  for (i = 0; i < n; i++)
  {
      a[i] += i;
  }
```

3. private 子句

private 子句表示它所列出的变量对于每个线程来说都是私有的。其具体格式如下：

```
private(list)
```

具体用法如下：

```
#pragma omp parallel for private(i, a)  /* 在该并行区中，变量 i 与 a 为每个线程私有 */
  for (i = 0; i < n; i++)
  {
      a = i + 1;
      printf("Thread %d has a value of a = %d for i = %d\n",
              omp_get_thread_num(), a, i);
  }
```

4. lastprivate 子句

lastprivate 子句是 private 子句的扩展，它不仅包含了 private 子句的功能，而且还要将循环的最后一次迭代之后变量的值复制给原始的变量。其具体格式如下：

```
lastprivate(list)
```

其具体用法如下：

```
#pragma omp parallel for private(i) lastprivate(a)  /* 变量 a 具有 lastprivate 属性 */
```

```
    for (i = 0; i < n; i++)
    {
        a = i + 1;
        printf("Thread %d has a value of a = %d for i = %d\n",
                omp_get_thread_num(), a, i);
    }
    printf("Value of a after parallel for: a = %d\n", a);
   /* 此处 a 的值为上面循环的最后一次迭代之后 a 的值 */
```

5. firstprivate 子句

firstprivate 子句也是 private 子句的扩展，它不仅包含了 private 子句的功能，而且当执行到该并行结构时用对应变量的原始值初始化该变量。其具体格式如下：

```
firstprivate(list)
```

下面给出一个有关 firstprivate 子句使用的实例：

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define TRUE  1
#define FALSE 0

int main()
{
   int *a;
   int n = 2, nthreads, vlen, indx, offset = 4, i, TID;
   int failed;

   omp_set_num_threads(3);

   indx = offset;
   #pragma omp parallel firstprivate(indx) shared(a, n, nthreads, failed)
   /* 变量 indx 具有 firstprivate 属性 */
   /* 该并行区中 indx 的初始值为该指导语句之前 indx 的值 */
   {
      #pragma omp single
      {
         nthreads = omp_get_num_threads();
         vlen = indx + n * nthreads;
         if ( (a = (int *) malloc(vlen * sizeof(int))) == NULL )
            failed = TRUE;
```

```
            else
                failed = FALSE;
        }
    }

    if ( failed == TRUE ) {
        printf("Fatal error: memory allocation for a failed vlen = % d\n", vlen);
        return( - 1);
    }
    else
    {
        printf("Diagnostics:\n");
        printf("nthreads  = % d\n", nthreads);
        printf("indx      = % d\n", indx);
        printf("n         = % d\n", n);
        printf("vlen      = % d\n", vlen);
    }

    for(i = 0; i < vlen; i++) a[i] = - i - 1;

    printf("Length of segment per thread is % d\n", n);
    printf("Offset for vector a is % d\n", indx);
    # pragma omp parallel default(none) firstprivate(indx) \
            private(i, TID) shared(n, a)
    {
        TID = omp_get_thread_num();
        indx += n * TID;
        for(i = indx; i < indx + n; i++)
           a[i] = TID + 1;
    }

    printf("After the parallel region:\n");
    for (i = 0; i < vlen; i++)
        printf("a[ % d] = % d\n", i, a[i]);

    free(a);

    return(0);
}
```

运行结果如下所示：

```
Diagnostics:
nthreads = 3
indx     = 4
n        = 2
vlen     = 10
Length of segment per thread is 2
Offset for vector a is 4
After the parallel region:
a[0] = -1
a[1] = -2
a[2] = -3
a[3] = -4
a[4] = 1
a[5] = 1
a[6] = 2
a[7] = 2
a[8] = 3
a[9] = 3
```

6. reduction 子句

reduction 子句使用指定的操作对其列表中出现的变量进行归约操作。最初，线程组中所有线程都将保留该列表中每个变量的私有备份。在该结构结束时，根据指定的操作对所有线程中的相应变量进行归约操作，并更新该变量的全局值。其格式如下：

```
reduction(operator :list)
```

下面给出一个有关 reduction 子句使用的实例：

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define TRUE   1
#define FALSE 0
#define SUM_INIT 0

int main()
{
   int i, n = 25;
   int sum, a[n];
   int ref = SUM_INIT + (n - 1) * n/2;

   omp_set_num_threads(3);

   for (i = 0; i < n; i++)
       a[i] = i;

   #pragma omp parallel
   {
       #pragma omp single
           printf("Number of threads is %d\n", omp_get_num_threads());
   }
```

```
    sum = SUM_INIT;
    printf("Value of sum prior to parallel region: %d\n", sum);
    #pragma omp parallel for default(none) shared(n, a) \
                reduction( + :sum)        /* 对变量 sum 进行规约求和操作 */
        for (i = 0; i < n; i++)
            sum += a[i];

    printf("Value of sum after parallel region: %d\n", sum);
    printf("Check results: sum = %d (should be %d)\n", sum, ref);

    return(0);
}
```

运行结果如下所示：

```
Number of threads is 3
Value of sum prior to parallel region: 0
Value of sum after parallel region: 300
Check results: sum = 300 (should be 300)
```

7. copyin 子句

copyin 子句用来为线程组中所有线程的 threadprivate 变量赋相同的值。其中，主线程中该变量的值被作为初始值。其格式如下：

```
copyin(list)
```

3.2.5　运行时库函数

运行时环境例程函数监控并影响线程和并行环境。锁例程函数支持 OpenMP 的锁机制。定时例程函数支持一个简易的定时器。omp. h 头文件给出了运行时库函数的原型定义。

1. 环境例程函数

1）omp_set_num_threads

当执行到没有用 num_threads 子句来指定线程数目的并行区时，可以调用该函数设置并行区中线程的数目。其格式如下：

```
void omp_set_num_threads(int num_threads);
```

2）omp_get_num_threads

返回线程组中线程的数目。其格式如下：

```
int omp_get_num_threads(void);
```

3）omp_get_max_threads

当没有用 num_threads 子句指定并行区中线程的数目时，调用该函数获得当前创建一个新的线程组所含线程的最大数目。其格式如下：

```
int omp_get_max_threads(void);
```

4）omp_get_thread_num

返回当前线程的标识号（线程号），其值范围为从 0 开始到线程组中线程数目减去 1。其格式如下：

```
int omp_get_thread_num(void);
```

5）omp_get_num_procs

返回程序允许的最大进程数。其格式如下：

```
int omp_get_num_procs(void);
```

6）omp_in_parallel

如果该函数调用包含于活动并行区，则返回 true；否则，返回 false。其格式如下：

```
int omp_in_parallel(void);
```

7）omp_set_dynamic

设置是否允许对线程数目进行动态调整。其格式如下：

```
void omp_set_dynamic(int dynamic_threads);
```

8）omp_get_dynamic

返回线程数目动态调整是否已开启。其格式如下：

```
int omp_get_dynamic(void);
```

9）omp_set_nested

设置是否支持嵌套并行。其格式如下：

```
void omp_set_nested(int nested);
```

10）omp_get_nested

返回嵌套并行是否已开启。其格式如下：

```
int omp_get_nested(void);
```

11）omp_set_schedule

设置运行时调度方式。其格式如下：

```
void omp_set_schedule(omp_sched_t kind, int modifier);
```

12）omp_get_schedule

当使用了运行时调度时，返回当前正在使用的调度方式。其格式如下：

```
void omp_get_schedule(omp_sched_t * kind, int * modifier);
```

13）omp_get_thread_limit

返回程序可用的最大线程数目。其格式如下：

```
int omp_get_thread_limit(void);
```

14）omp_set_max_active_levels

设置嵌套的活动并行区的最大数目。其格式如下：

```
void omp_set_max_active_levels(int max_levels);
```

15）omp_get_max_active_levels

返回嵌套的活动并行区的最大数目。其格式如下：

```
int omp_get_max_active_levels(void);
```

16）omp_get_level

返回执行该函数调用嵌套的并行区的数目。其格式如下：

```
int omp_get_level(void);
```

17）omp_get_ancestor_thread_num

针对给定的当前线程的嵌套级别,返回父线程或当前线程的线程号。其格式如下：

```
int omp_get_ancestor_thread_num(int level);
```

18）omp_get_team_size

针对给定的当前线程的嵌套级别,返回父线程或当前线程所属的线程组的大小。其格式如下：

```
int omp_get_team_size(int level);
```

19）omp_get_active_level

如果该函数调用包含于活动并行区,则返回活动并行区的嵌套级别。其格式如下：

```
int omp_get_active_level(void);
```

下面举一个实例展示环境变量的相关使用：

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <omp.h>
#define TRUE  1
#define FALSE 0

int main()
{
   int TID = -1;

#ifdef _OPENMP
   (void) omp_set_dynamic(FALSE);
   if (omp_get_dynamic())
    {printf("Warning: dynamic adjustment of threads has been set\n");}
   (void) omp_set_num_threads(3);
   (void) omp_set_nested(TRUE);
   if (! omp_get_nested()) {printf("Warning: nested parallelism not set\n");}
#endif

   printf("Nested parallelism is %s\n",
            omp_get_nested() ? "supported" : "not supported");

#pragma omp parallel private(TID)
   {
     TID = omp_get_thread_num();
     printf("Thread %d executes the outer parallel region\n", TID);

     #pragma omp parallel num_threads(2) firstprivate(TID)
     {
        printf("TID %d: Thread %d executes inner parallel region\n",
               TID, omp_get_thread_num());
     }
   }

   return(0);
}
```

运行结果如下所示：

```
Nested parallelism is supported
Thread 2 executes the outer parallel region
Thread 1 executes the outer parallel region
TID 2: Thread 0 executes inner parallel region
TID 2: Thread 1 executes inner parallel region
TID 1: Thread 0 executes inner parallel region
TID 1: Thread 1 executes inner parallel region
Thread 0 executes the outer parallel region
TID 0: Thread 0 executes inner parallel region
TID 0: Thread 1 executes inner parallel region
```

2. 锁例程函数

1）omp_init_lock 与 omp_init_nest_lock

初始化锁。其格式如下：

```
void omp_init_lock(omp_lock_t * lock);
void omp_init_nest_lock(omp_nest_lock_t * lock);
```

2）omp_destroy_lock 与 omp_destroy_nest_lock

销毁锁。其格式如下：

```
void omp_destroy_lock(omp_lock_t * lock);
void omp_destroy_nest_lock(omp_nest_lock_t * lock);
```

3）omp_set_lock 与 omp_set_nest_lock

加锁。其格式如下：

```
void omp_set_lock(omp_lock_t * lock);
void omp_set_nest_lock(omp_nest_lock_t * lock);
```

4）omp_unset_lock，omp_unset_nest_lock

解锁。其格式如下：

```
void omp_unset_lock(omp_lock_t * lock);
void omp_unset_nest_lock(omp_nest_lock_t * lock);
```

5）omp_test_lock，omp_test_nest_lock

在不挂起正在执行的任务的情况下加锁。其格式如下：

```
int omp_test_lock(omp_lock_t * lock);
int omp_test_nest_lock(omp_nest_lock_t * lock);
```

3. 定时例程函数

1）omp_get_wtime

以秒为单位返回消耗的时间。其格式如下：

```
double omp_get_wtime(void);
```

2）omp_get_wtick

返回 omp_get_wtime 使用的计时器的精度。其格式如下：

```
double omp_get_wtick(void);
```

3.2.6 环境变量

环境变量中的字母均为大写字母，赋给它们的值则不区分大小写。

1. OMP_SCHEDULE type[，chunk]

设置运行时调度类型与调度块的大小。合法的 OpenMP 调度类型包括 static、dynamic、guided 或 auto，调度块的大小是一个正整数。相关含义与使用请参考相关的文档与书籍。

2. OMP_NUM_THREADS num

设置并行区中的线程数目。

3. OMP_DYNAMIC dynamic

设置并行区中的线程数目是否可以调整。

4. OMP_NESTED nested

设置是否支持嵌套并行。

5. OMP_STACKSIZE size

设置 OpenMP 线程使用的堆栈大小。堆栈大小的合法值 X(一个正整数)包括 X、XB、XK、XM、XG，如果没有设置 B、K、M 或者 G 等单位。默认的单位是 kilobytes(K)。

6. OMP_WAIT_POLICY policy

设置等待进程的行为控制策略。其合法值包括 active(进程等待时会消耗 cpu 周期)和 passive。

7. OMP_MAX_ACTIVE_LEVELS levels

设置活动的嵌套并行区的最大数目。

8. OMP_THREAD_LIMIT limit

设置 OpenMP 程序使用的最大线程数目。

3.2.7 运行及调试

1. gdb

在 Linux 软件开发中，GDB 是一个功能强大、运行稳定的程序调试工具。GDB 不仅

可以调试单进程的程序，也可以调试多进程与多线程的程序。

GDB 提供了一些命令，可以跟踪程序的执行。在编译程序时，需要加上编译选项-g，以便用户使用 GDB 调试程序。

2. Totalview

TotalView Debugger 是一款由 TotalView Technologies 公司开发的产品，它拥有最广泛和最完善的 C/C++ 调试功能。它提供了高可靠性、先进和丰富的功能以及便捷的使用，使之成为 UNIX 和 Linux 平台下最好的调试器。它支持 MPI、多线程、OpenMPI、C/C++、Fortran 等混合语言编程。

3.2.8 OpenMP 编程实例

下面给出计算 pi 的相关程序，供大家参考。

1. C 语言写的串行程序

```
#include <stdio.h>
#include <time.h>
void main ()
{
    static long num_steps = 1000000000;
    double step;
    int i;
    double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    for (i = 0;i < num_steps; i++)
    {
        x = (i + 0.5) * step;
        sum = sum + 4.0/(1.0 + x * x);
    }
    pi = step * sum;
    printf("Pi = %21.20f ( %d steps)\n",pi,num_steps);
}
```

运行结果如下所示：

```
Pi = 3.141592653589763 (1000000 steps), 50000 ms
```

2. 使用共享任务结构并行化后的程序

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include<omp.h>
#define NUM_THREADS 4
void main()
{
    static long num_steps = 1000000000;
    double step;
    int i;
    double x, pi, sum[NUM_THREADS];
    step = 1.0/(double)num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        double x;
        int id;
        id = omp_get_thread_num();
        sum[id] = 0.0;
     #pragma omp for
            for(i = id;i < num_steps;i++){
              x = (i + 0.5) * step;
              sum[id] += 4.0/(1.0 + x * x);
            }
    }

    for(i = 0, pi = 0.0; i < NUM_THREADS; i++)
             pi += sum[i] * step;

    printf("Pi = %21.20f (%d steps)\n",pi,num_steps);
}
```

运行结果如下所示：

```
Pi = 3.14149740762628182367 (100000 steps), 10000 ms
```

3. 使用归约后的并行程序

```
#include <stdio.h>
#include <time.h>
#include <omp.h>
#define NUM_THREADS 4
void main ()
{
    time_t start_time, finish_time;
    static long num_steps = 1000000;
    double step;
    int i;
```

```
    double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    start_time = clock();
    #pragma omp parallel for reduction( + :sum) private(x)
    for (i = 0;i < num_steps; i++)
    {
        x = (i + 0.5) * step;
        sum = sum + 4.0/(1.0 + x * x);
    }

    pi = step * sum;
    finish_time = clock();
    printf("Pi = %21.20f ( %d steps)\n",pi,num_steps);
}
```

运行结果如下所示：

```
Pi = 3.14159265358987394023 (1000000 steps), 10000 ms
```

读者可尝试运行本节的实例程序。如果有兴趣可以加入 clock()时间函数，看一下每个程序的运行所用时间。虽然上述程序都非常简短，但却可以很好地理解 OpenMP。如果对 OpenMP 及其编程感兴趣的话，可以查找相关文档与书籍进行深入的学习。

本节小结

OpenMP 是一种基于共享存储系统的并行编程模型。与 MPI 和 PVM 相比，OpenMP 提供了更为简单的编程模型，更易于编程，但其系统开销比 MPI 大。同时，OpenMP 实现并不要求编译器检查数据相关性、访问冲突、死锁、竞争或其他可能导致程序错误执行的问题，它完全依赖于用户来保证编译指导的正确性。

3.3　MapReduce

并行计算的一个重要发展是基于机群计算的大量增加，其典型代表是 Google 公司开发的 MapReduce 并行编程模型。MapReduce 是对庞大数据档案进行搜索的工具。Google 公司使用 MapReduce 完成诸如 PageRank 的操作。现在，MapReduce 的应用范围非常广泛，Google、FaceBook 和 Amazon 等诸多国内外知名大公司都在使用 MapReduce 加速或者简化各自公司的业务。

3.3.1　MapReduce 简介

MapReduce 是 Google 公司提出的并行编程模型，其特点是简单易学、适用广泛，能

够降低并行编程难度，让程序员从繁杂的工作中解脱出来，可以轻松地编写简单、高效的分布式并行程序，目前已被 Google 用在处理海量数据的实际工作中。

MapReduce 并行编程模型的最大优势在于能够有效降低并行编程的复杂度，提高编程效率。其主要贡献在于：

① 使用廉价的商用机器组成机群，费用较低，同时又具有较高的性能。那些原本需要使用大型专用设备才能完成的事情，现在利用普通的 PC 机群就能完成。

② 松耦合和无共享结构使之具有良好的可扩展性。MapReduce 的一个重要应用领域就是云计算，云计算要求良好的可扩展性，MapReduce 很好地满足了这一点。

③ 提供了一个运行时支持库，它支持任务的自动并行执行。提供的接口便于用户高效地进行任务调度、负载均衡、容错和一致性管理等。用户只要根据需要自定义 Map、Reduce 和 Partition 等函数即可。

④ MapReduce 的使用范围广泛。目前，Google 等诸多国内外知名大公司都在使用 MapReduce 来加速或者简化各自公司的业务。MapReduce 还广泛应用于云计算、数据挖掘、图像处理等领域。随着科技的进步，MapReduce 并行编程模型将会更广泛地应用。

3.3.2 MapReduce 实例

1. 单词计数

Google 公司提出 MapReduce 并行编程模型，简化了其分布式系统的编程。应用程序开发人员只需关注应用本身，而机群的可靠性、可扩展性等问题，则交由平台处理。MapReduce 提供了简单但强大的接口 Map 和 Reduce，通过这两个接口，MapReduce 并行编程模型可自动并发和分布执行大规模的计算。Map 和 Reduce 的概念来自于更早的 LISP 等函数式语言。Map 操作处理输入数据的每个逻辑“记录”，产生一组中间的(key, value)键值对。接下来，在所有相同 key 的中间结果上执行 Reduce 操作，来合并适当的数据，继而得到下一步的输入数据或计算结果。

下面，以统计文档集合中每个单词出现的次数(即单词计数)为例来说明 MapReduce 的过程。其伪代码如下：

```
Map(String key, String value):
    /* key: 文档的名字
    /* value: 文档的内容 */
    for each word w in value:
        EmitIntermediate(w, "1");
Reduce(String key, Iterator values):
    /* key: 一个词
    /* values: 一个计数列表 */
    int result = 0;
```

```
for each v in values:
    result += ParseInt(v);
Emit(AsString(resut));
```

其中，Map 函数生成每个词和该词的出现次数(在该例中就是 1)。Reduce 函数把生成的每个特定的词的出现次数累加在一起。另外，用户使用输入输出文件的名字和可选的调节参数来填充一个 MapReduce 对象。然后，调用 MapReduce 函数，并把对象传递给它，将用户的代码和 MapReduce 库链接在一起。

该例用字符串作为输入和输出。从概念上讲，Map()和 Reduce()函数的输入/输入类型应遵循以下规则：

```
Map(k1, v1)            →    list(k2, v2)
Reduce(k2, list(v2))   →    list(v2)
```

也就是说，输入的键值和输出的键值属于不同的域，中间的键值和输出的键值属于相同的域。

2. 其他实例

MapReduce 并行编程模型在实际系统中使用非常广泛，下面列举一些可以利用 MapReduce 并行编程模型进行计算的例子。

1) 分布式的 Grep(UNIX 工具程序，可查找文件内的字符串)

- 如果输入行匹配给定样式，Map 函数就输出该行。
- Reduce 函数将中间数据复制给输出。

2) 计算 URL 的访问频率

- Map 函数处理 Web 页面请求的记录，输出(URL, 1)。
- Reduce 函数把相同 URL 的 value 值累加起来，产生一个(URL, 记录总数)对。

3) 逆向 Web-Link 图

- Map 函数为每个链接输出(目标 URL, 源 URL)对。其中，一个 URL 叫做目标，包含该 URL 的页面叫做源。
- Reduce 函数聚合所有与同一目标 URL 相关联的源 URL 列表，产生(目标 URL, list(源 URL))对。

4) 主机术语向量指标(Term-Vector per Hosts)

术语向量用来概述出现在一个文档或一个文档集中最重要的一些词，用(词, 频率)对表示。

- Map 函数为每一个输入文档(主机名是从文档的 URL 取的)产生一个(主机名, 术语向量)对。
- Reduce 函数接收给定主机的所有文档的术语向量。它把这些术语向量加在一

起，丢弃低频的术语，然后产生一个最终的(主机名，术语向量)对。

5) 倒排索引

- Map 函数分析每个文档，然后产生一个(词，文档号)序列对。
- Reduce 函数处理指定词的所有序列对，排序相应的文档 IDs，并产生一个(词，list(文档 ID))对。所有的输出对，组成一个简单的倒排索引。它可以简单增加跟踪词位置的计算。

6) 分布式排序

- Map 函数从每个记录中提取关键字，并且产生一个(关键字，记录)对。
- Reduce 函数不改变任何的对。

3.3.3 MapReduce 基本原理介绍

1. 执行方式

Google 公司最早提出了 MapReduce 并行编程模型。当用户程序调用 MapReduce 运行支持库时，通过调用 Map 函数，输入数据被自动分割成 M 片分布到多台机器上，输入的片能够在不同的机器上并行处理。通过调用 Reduce 函数，中间 key 被 Partition 函数分割，从而形成 R 片(例如，hash(key) mod R)，它们也分布到多台机器上。分割数量 R 和 Partition 函数由用户来指定。

图 3-28 显示了 Google 公司实现的 MapReduce 操作的全过程(如参考文献[2]所示)。当用户的程序调用 MapReduce 的时候，将发生下面一系列动作(下面的数字与图 3-28 中的数字标签相对应)：

① 首先，用户程序将输入数据分割成 M 份，每份为 16MB 到 64MB 大小的数据块(可通过参数设定数据块的大小)。然后，开始在集群上拷贝程序。这些程序拷贝中有一

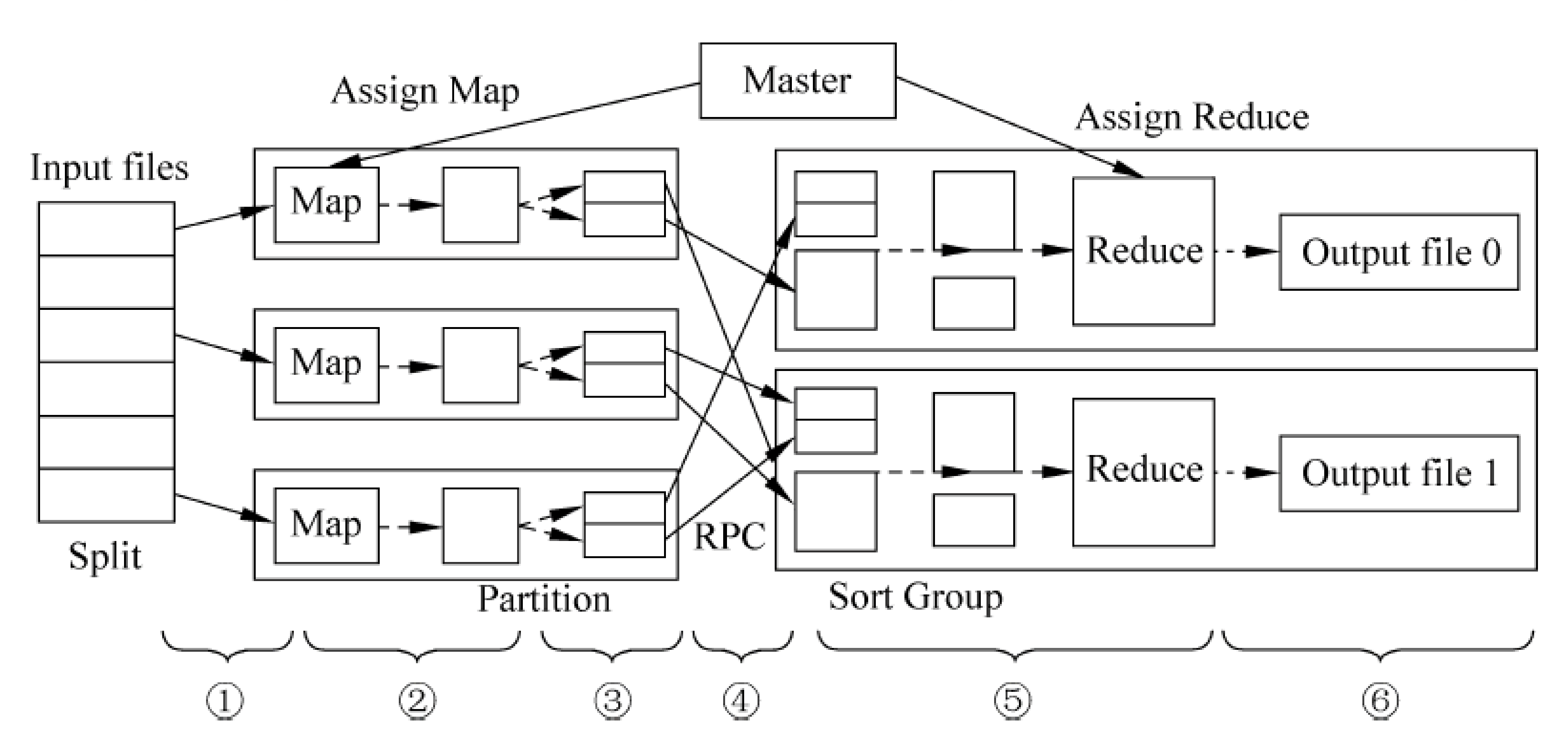


图 3-28 MapReduce 的执行过程

份是 Master，其余都是向 Master 请求任务的 Worker。

② 一旦分配到 Map 任务，Worker 便从相应的输入数据中分析出(key，value)对，并把每个(key，value)对作为用户定义的 Map 函数的输入。Map 函数产生的中间值(key，value)对被存储在内存中。

③ 存储在内存里的中间值(key，value)对会被定期写入本地磁盘中，用户定义的 Partition 函数将其划分为多个部分。Master 负责将这些中间值(key，value)对在本地磁盘上的存储位置传送给执行 Reduce 任务的 Worker。

④ 当一个执行 Reduce 任务的 Worker 得到 Master 的位置通知时，它使用远程过程调用从执行 Map 任务的 Worker 的本地磁盘中读取中间值(key，value)对。

⑤ 当一个执行 Reduce 任务的 Worker 从远程读取到所有所需的中间值(key，value)对之后，通过排序使具有相同 key 的内容聚合在一起。由于有许多不同的 key 映射到相同的 Reduce 任务，所以排序是必须的。如果中间数据比内存还大，那么还需要进行外部排序。之后形成(key，list(value))对，将其作为 Reduce 函数的输入。

⑥ 将每个 Reduce 函数的输出分别放到相应的输出文件中。当所有的 Map 和 Reduce 任务都执行完毕，Master 唤醒用户程序。此时，用户程序中的 MapReduce 调用返回到用户代码中。

作为一个目前被广泛使用的并行编程模型，MapReduce 可以处理 TB 或者 PB 量级的数据，它能非常方便地在许多机器上实现海量数据的并行处理。

2. MapReduce 语义

虽然，Google 公司提出的 MapReduce 并行编程模型可以用最简单的 Map 和 Reduce 这两个函数满足部分应用的需求，但随着该并行编程模型的发展，产生了 MapReduce 的一些扩展语义，如下所示。

① Split 函数：由于高性能计算问题的输入是多种多样的，很难找到一种固定的输入模式，只要可以将这种输入模式转换成相应的(key，value)对形式即可。因此，Split 函数的输入可以是用户定义的任何结构的输入，如 WordCount(单词计数)中的单词文本。除输入数据外，用户可能需要提供 Split 粒度、key 和 value 的生成过程的额外定义。之后，用户只需将 key 和 value 作为参数传递给系统提供的接口函数，系统便会自动生成 Split，并将其缓存入队列。

② Map 函数：Map 函数是由用户定义的，例如 Map(key_type key，value_type value)。其中的 key 和 value 分别由用户在 Split 阶段定义。Map 函数应当实现对 key 和 value 存储的数据进行操作并产生中间值(key，value)对。

③ Combine 函数：如果出于对性能和效率的考虑，Combine 函数将 Map 操作的输出中间值(key，value)对收集到一些 list 中，一个 key 值对应一个 list。当写入一定数量的

(key, value)对时,这部分 list 就会被处理并产生(key, list(value))对。

④ Partition 函数:用户自己定义 Partition 函数,该函数把 Combine 函数产生的(key, list(value))对缓存到不同的 Partition 块中,每个 Partition 块都是一个数组结构。Partition 函数典型的实现是 key mod n,n 是本地 Partition 块的数量,Partition 函数的返回值就是本地 Partition 的编号,其范围为从 0 到 $n-1$。

⑤ Sort 函数:当一个执行 Reduce 任务的 Worker 从远程读取到所有所需的中间值(key, list(value))对之后,将它们写入一个文件中。通过 Sort 函数对该文件进行排序,把具有相同 key 值的(key, list(value))对排列在一起,之后将该文件作为 Reduce 函数的输入。

⑥ Reduce 函数:用户自己定义 Reduce 函数,该函数顺序读入输入文件,将一个 key 对应的 value 或者 list(value)传送给 Reduce 函数。完成操作之后,再去读取下一个 key。每个 Reduce 任务最后得到一个输出文件。

⑦ Merge 函数:Merge 函数把每个 Reduce 任务的输出文件聚合到一起,得到最后的输出文件。

3. Hadoop 的 MapReduce

受 Google 公司提出的 MapReduce 并行编程模型的启发,Hadoop(源于 Apache 的 Lucene 项目)应运而生,目前它是一个开源的、可运行于大型分布式集群上的、被广泛使用的并行编程框架。

Hadoop 提供了一个支持 MapReduce 并行编程模型的部件,使用分布式文件系统 HDFS(Hadoop Distributed Filesystem)代替 Google 的 GFS,并且基于 Google 的 Big Table 开发了自己的分布式数据库 HBase。

Hadoop 系统的结构如图 3-29 所示,它有一个 Master 和多个 Slave。其中,Master 主要负责 NameNode 和 JobTracker 的工作,JobTracker 的主要职责是启动、跟踪和调度各个 Slave 的任务执行。每个 Slave 主要负责 DataNode 和 TaskTracker 的工作,TaskTracker 能够根据应用的要求对本地数据执行 Map 与 Reduce 任务。

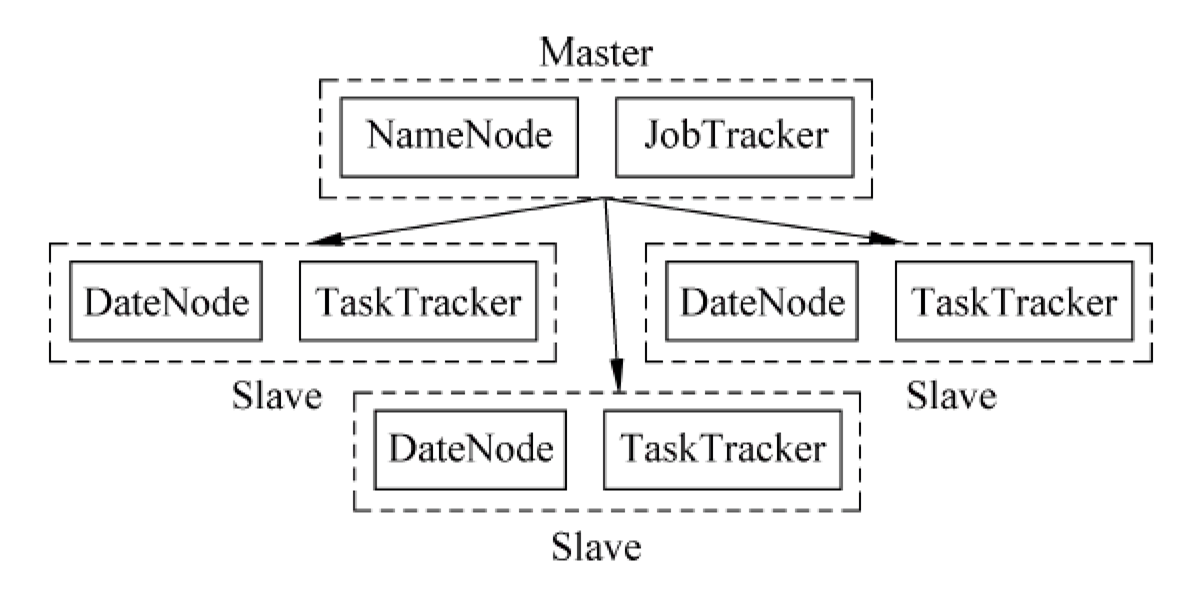


图 3-29 Hadoop 的结构示意图

Hadoop 中 MapReduce 的执行步骤如下所示：

① 在执行 MapReduce 任务时，首先通过 FileSplit 把一个输入数据集分割成相互独立的数据块，这些数据块作为 Map 任务的输入。然后，将程序复制到每个结点上。在这些结点中，有一个是 JobTracker 结点，其他结点都是 TaskTracker 结点。JobTracker 结点负责 TaskTracker 结点群的任务调度和分配。Hadoop Job client 负责向 JobTracker 结点提交任务，并且配置 JobTracker 结点的参数。

② 被分配到 Map 任务的 TaskTracker 结点读取相应的输入内容，从中分析出(key, value)对，调用用户定义的 Map 函数产生中间值(key, value)对。

③ 当 Map 操作输出它的键值(key, value)对时，出于对性能和效率的考虑，Hadoop 会提供一个合成器(Combine)。通过 Combine 操作，Map 操作所产生的键值(key, value)对就不会马上写入输出文件，它们会被收集在一些 list 中，一个 key 对应一个 list。当写入一定数量的键值(key, value)对时，这部分的 list 会被 Combine 处理，产生中间值(key, list(value))对。

④ 当一个 Reduce 任务开始时，其输入分布在各个结点上 Map 任务的输出文件中。在执行 Reduce 任务的 Worker 从远程读取到所有所需的中间值(key, list(value))对之后，必须通过排序使具有相同 key 的中间值(key, list(value))对聚合在一起，并作为 Reduce 函数的输入。最后的输出结果由每个 Reduce 的输出文件组成。

Hadoop 也存在一些不足：

- Hadoop 主要针对大块的数据文件(如 GB、TB 级别)，由于系统开销等原因，小块数据的处理速度并不一定比串行程序快；
- 由于中间结果文件被存储在各个分布式计算结点的内存或磁盘上的，如果计算产生的中间结果文件非常巨大的话，则 Reduce 过程需要通过远程过程调用获取这些中间结果文件，会加大网络传输的开销；
- Hadoop 作为一个比较新的项目，性能和稳定性的提高还需要一定的时间。

3.3.4　容错

由于 MapReduce 的设计初衷是用于在成百、上千台机器上进行海量数据的处理，所以该平台必须考虑机器发生故障时的容错处理。下面，简单介绍一下 MapReduce 的容错处理机制。

如图 3-28 所示，Master 结点周期性地 ping 每个 Worker 结点。如果在一个时间段内 Worker 结点没有返回信息，Master 结点就将标注该 Worker 结点失效。由该失效 Worker 完成的所有 Map 任务被重新设置成初始空闲状态，并分配给其他 Worker 执行。同样地，每个在失效 Worker 上正在运行的 Map 或 Reduce 任务，也将被重新设置成空闲状态，并被重新调度。在一个失效机器上已经完成的 Map 任务将被再次执行(因为其输

出存储在它的磁盘上，所以不可访问），而已经完成的 Reduce 任务将不会被再次执行（因为其输出存储在全局文件系统中）。当一个 Map 任务首先被 Worker A 执行之后，又被 Worker B 执行（因为 Worker A 失效了），则该 Map 任务被重新执行的消息将通知给所有执行 Reduce 任务的 Worker。未从 Worker A 读取数据执行 Reduce 任务的任何 Worker 都将从 Worker B 读取数据。

MapReduce 运行时系统会定期设定 Master 检查点。如果 Master 任务失效，则可从上次设定的最后一个检查点开始启动另一个 Master 结点。然而，由于只有一个 Master 结点，如果 Master 结点处发生故障，则中止计算。客户可以检查最后一个 Master 检查点，并且可以根据需要重新执行 MapReduce 操作。

Map 和 Reduce 任务的可靠性是由输出进行原子提交完成的。每个正在进行的任务将输出写入一个私有的临时文件中。当全部写完之后，进行提交操作，并将这些临时文件变为永久保存的文件。

3.3.5 MapReduce 编程实例、运行与分析

下面给出在 Hadoop 开源云计算包中关于 WordCount（单词计数）例子的实现，在其中调用了 Hadoop 相关的 API。

在该实例中，首先读取输入文件，把每行分割成一个一个的单词，之后对这些单词进行计数，最后输出排好序的单词列表和每个单词出现的次数。其具体代码如下所示：

```
package org.apache.hadoop.mapred;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.StringTokenizer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reducer;
```

```
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
public class WordCount extends Configured implements Tool
{
    /* MapClass 类统计每一行中的单词个数
     * 把每一行的输入分割成一个一个单词,并且将它们以(<b>word</b>, <b>1</b>)
     * 的形式提交   */
    public static class MapClass extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, IntWritable>
    {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();
        public void map(LongWritable key, Text value,
            OutputCollector<Text, IntWritable> output,
            Reporter reporter) throws IOException
        {
            String line = value.toString();
            StringTokenizer itr = new StringTokenizer(line);
            while (itr.hasMoreTokens())
            {
                word.set(itr.nextToken());
                output.collect(word, one);
            }
        }
    }

    /* Reduce 类,提交输入值的和 */
    public static class Reduce extends MapReduceBase
        implements Reducer<Text, IntWritable, Text, IntWritable>
    {
        public void reduce(Text key, Iterator<IntWritable> values,
            OutputCollector<Text, IntWritable> output,
            Reporter reporter) throws IOException
        {
            int sum = 0;
            while (values.hasNext())
            {
                sum += values.next().get();
            }
            output.collect(key, new IntWritable(sum));
        }
}
```

```
static int printUsage()
{
    System.out.println("wordcount [-m <maps>] [-r <reduces>] <input> <output>");
    ToolRunner.printGenericCommandUsage(System.out);
    return -1;
}

/* word count map/reduce 运行时系统主程序
 * 包括提交 Map/Reduce 任务
 * 当 job tracker 发现通信出现问题时,抛出 IOException */
public int run(String[] args) throws Exception
{
    JobConf conf = new JobConf(getConf(), WordCount.class);
    conf.setJobName("wordcount");
    // the keys are words (strings)
    conf.setOutputKeyClass(Text.class);
    // the values are counts (ints)
    conf.setOutputValueClass(IntWritable.class);
    conf.setMapperClass(MapClass.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);
    List<String> other_args = new ArrayList<String>();
    for(int i = 0; i < args.length; ++i)
    {
        try
        {
            if ("-m".equals(args[i]))
            {
                conf.setNumMapTasks(Integer.parseInt(args[++i]));
            }
            else if ("-r".equals(args[i]))
            {
                conf.setNumReduceTasks(Integer.parseInt(args[++i]));
            }
            else
            {
                other_args.add(args[i]);
            }
        }
        catch (NumberFormatException except)
        {
            System.out.println("ERROR: Integer expected instead of " + args[i]);
            return printUsage();
        }
```

```
            catch (ArrayIndexOutOfBoundsException except)
            {
                System.out.println("ERROR: Required parameter missing from " +
                                  args[i-1]);
                return printUsage();
            }
        }
        if (other_args.size() != 2)
        {
             System.out.println("ERROR: Wrong number of parameters: " +
                                other_args.size() + " instead of 2.");
             return printUsage();
        }
        FileInputFormat.setInputPaths(conf, other_args.get(0));
        FileOutputFormat.setOutputPath(conf, new Path(other_args.get(1)));
        JobClient.runJob(conf);
        return 0;
    }

    public static void main(String[] args) throws Exception
    {
        int res = ToolRunner.run(new Configuration(), new WordCount(), args);
        System.exit(res);
    }
}
```

下面是 WordCount(单词计数)的运行样例及其结果。

输入样例：

```
$ bin/hadoop fs -ls /user/joe/wordcount/input/
/user/joe/wordcount/input/file01
/user/joe/wordcount/input/file02
$ bin/hadoop fs -cat /user/joe/wordcount/input/file01
Hello World Bye World
$ bin/hadoop fs -cat /user/joe/wordcount/input/file02
Hello Hadoop Goodbye Hadoop
```

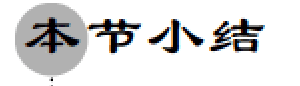
运行程序：

```
$ bin/hadoop jar /user/joe/wordcount.jar org.myorg.WordCount
/user/joe/wordcount/input /user/joe/wordcount/output
```

输出：

```
$ bin/hadoop fs -cat /user/joe/wordcount/output/part-r-00000
```

```
Bye 1
Goodbye 1
Hadoop 2
Hello 2
World 2
```



本节小结

目前,MapReduce 并行编程模型正改变着海量数据的并行计算方式,必将在并行计算领域发挥越来越重要的作用。MapReduce 应用广泛、创新独特,它使得以前只能在大型专用硬件上处理的事情(如处理千兆级别的数据),现在在普通的 PC 集群上即可进行。在亚马逊的 Amazon Elastic MapReduce 产品中,以 Web 服务的方式很好地应用了 Apache Hadoop (MapReduce 的一种实现)。而且,MapReduce 还被集成到 IBM、Oracle 等公司的一些主流解决方案中,并被广泛应用于云计算服务器。

3.4 CUDA

2007 年 6 月,NVIDIA 公司推出了通用计算产品 CUDA(Compute Unified Device Architechture),其目前的版本为 4.0。CUDA 是一种将 GPU 作为数据并行计算设备的软硬件体系。与传统的 GPU 通用计算开发方式相比,使用 CUDA 进行编程更简单,功能也更强大,应用领域更广泛,支持 CUDA 的硬件性能更强。CUDA 完全改变了 PC 中的一些规则,使用 GPU 进行计算无须额外的成本开销,却可以在一些应用中至少获得一个数量级的加速。在科学计算中,一些过去必须由集群处理的任务,现在也可由桌面 PC 完成了,这使得科研人员能够更关注于研究本身。目前,CUDA 架构已应用于 GeForce、ION、Quadro 及 Tesla GPU 上。

目前,很多重要的消费级视频应用程序都已使用或即将使用 CUDA 进行加速,其中不乏 Elemental Technologies、MotionDSP 以及 LoiLo 等公司的产品。在科研界,CUDA 自出现以来一直受到热捧。

例如,CUDA 能够对 AMBER 进行加速。AMBER 是一款分子动力学模拟程序,在全世界的学术界与制药企业中有超过 60 000 名研究人员使用该程序加速新药的探索工作。在金融市场,Numerix(为近 400 家金融机构广泛使用)和 CompatibL 使用 CUDA 实现了一款全新的对手风险应用程序 18 倍的性能提升。

CUDA 的广泛应用造就了专用于计算的 Tesla GPU 的崛起。全球财富五百强企业现已经安装了 700 多个 GPU 集群,这些企业涉及各个领域,例如能源领域的斯伦贝谢与雪佛龙,银行业的法国巴黎银行等。在这些 GPU 集群中,GPU 将不仅仅是图形处理器,

它还被作为所有应用程序均可使用的通用并行处理器。

3.4.1　简介

在 GPU 支持通用计算之前，程序员通过图形 API 进行编程，这些 API 主要应用于图像和多媒体领域，比如 DirectX 和 OpenGL。由于 GPU 实现通用计算存在着许多困难(比如，通用图形 API 的编程复杂，难于学习，一些程序会遇到瓶颈；另外，GPU 编程缺乏灵活性，大大限制了 GPU 性能的发挥)，所以 AMD 和 NVIDIA 公司分别推出了面向 GPGPU(通用处理 GPU)的编程语言：Brook＋和 CUDA。

Brook＋是基于斯坦福大学的 Brook 提出的，它和 Brook 一样也是一种类 C 的编程语言，它将“流”概念引入 GPU 计算模型，从而支持 GPGPU。Brook＋中的“流”被定义成一个可以被并行操作的数组。同时，定义 Kernel，它是作用在流上的函数。

而 CUDA 却将 GPU 上的计算与图形完全区分开，它通过关键字指派数据并行函数及其相关数据以多线程的方式在 GPU 上运行，这更类似于传统的多线程编程，CUDA 也要求程序员对 GPGPU 的体系结构及其存储层次有一定的了解。

从图 3-30 可看出 NVIDIA 公司在 GPGPU 方面所做的贡献，它使编程模式从单纯的 CPU 编程逐渐走向 GPU 的 CUDA 编程。本节的主要内容是介绍 NVIDIA 公司的 CUDA 系列产品。

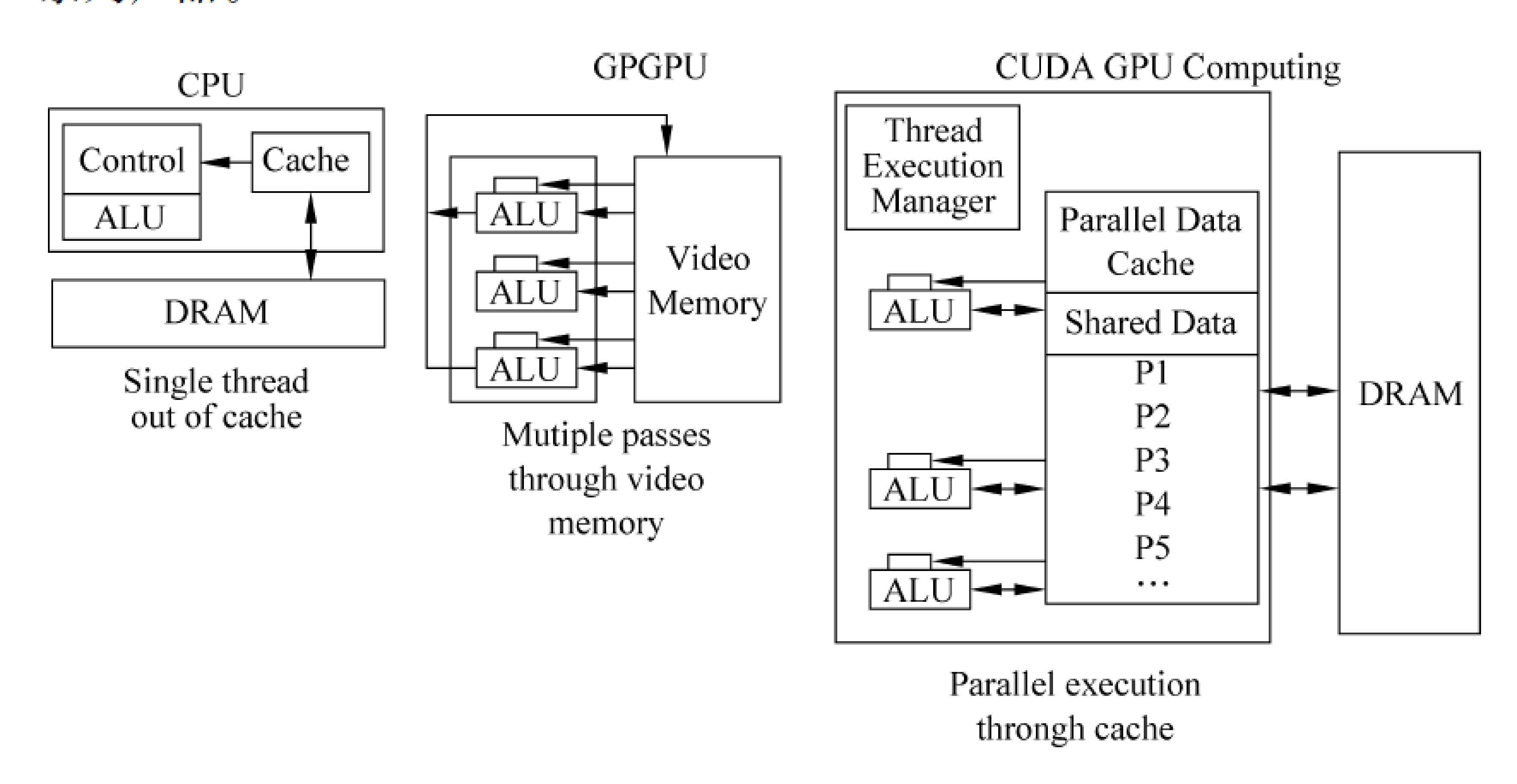


图 3-30　编程模式的演进

CUDA 使用的软件堆栈由以下三层构成：CUDA Library、CUDA runtime API、CUDA driver API。CUDA 的核心是 CUDA C 语言，它包含对 C 语言的最小扩展集和一个运行时库。图 3-31 为 CUDA 支持的语言和编程层次结构。

GPU Computing Application				
C/C++	OpenCL	DirectX Computer	Fortran	Java and Python
NVIDIA GPU with the CUDA Parallel Computing Architecture				

图 3-31 CUDA 编程语言及其层次结构

以 CUDA C 为例，它主要提供了：

- CUDA C 语言及对应的编译器　CUDA C 其实就是 C 语言的变种，它支持四大特性，即可定义程序中运行在 GPU 或 CPU 上的部分，可定义位于 GPU 中的变量存储类型，利用 Kernel、block、grid 定义并行计算，State 变量。
- CUDA 库　包含了很多有用的数学应用，如 CUFFT、CULIBS 等。
- CUDA runtime　其实就是 JIT(Just In Time)编译器，动态地将 PTX 中间代码编译成符合实际平台的硬件代码，并进行特定的优化。
- CUDA driver　是相应 API 与 GPU 打交道的接口。

图 3-32 为 CUDA 的软件体系结构示意图。

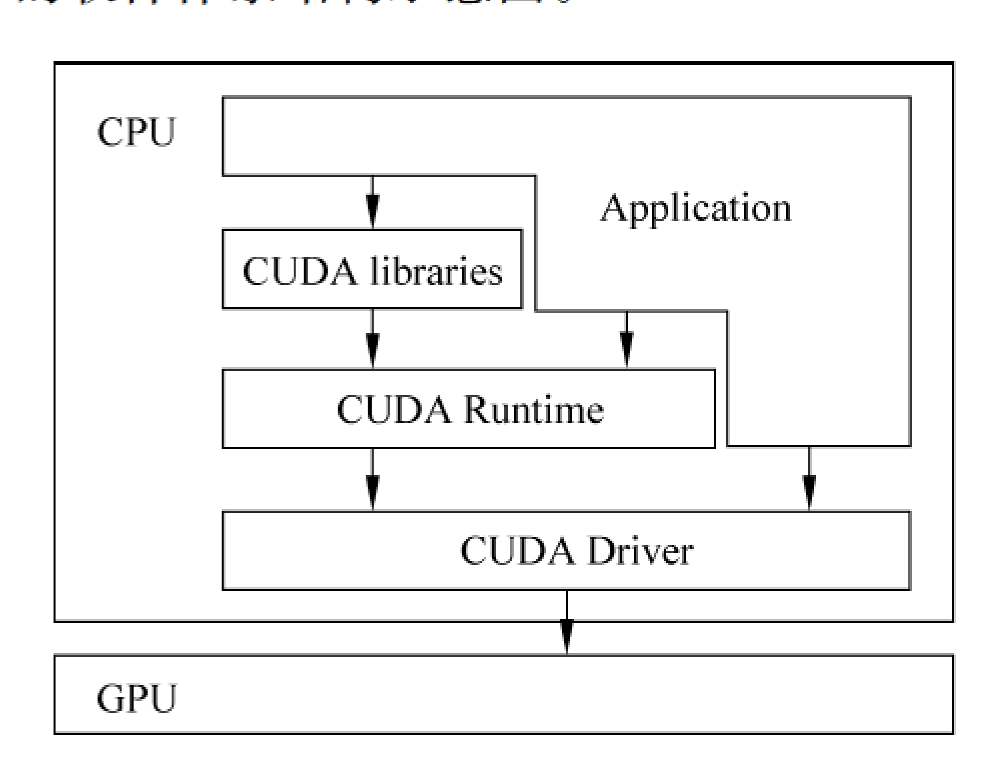


图 3-32 CUDA 的软件结构示意图

CUDA 的软件组成包括硬件驱动、CUDA 驱动 API、运行时 API、两个通用算术库 CUFFT 和 CUBLAS。CUDA 改进了 DRAM 的读写灵活性，使 GPU 与 CPU 的机制相吻合。另一方面，CUDA 提供了片上共享内存，使线程之间可以共享数据，应用程序可以利用共享内存来减少通过 DRAM 进行的数据传送，降低对 DRAM 内存带宽的依赖。

3.4.2 CUDA 的安装和配置

1. 安装 CUDA 相关组件

NVIDIA Parallel Nsight Toolkit 是目前 NVIDIA 公司最新提供的 CUDA 系列工具

(可从 http://developer.nvidia.com 处下载),支持 Windows(32 位与 64 位版本)和许多不同的 Linux 版本。

如何获得一款支持 CUDA 的 GPU 或系统呢?

- 想了解面向高性能计算和超级计算应用程序的 Tesla 的相关信息,请访问:http://www.nvidia.cn/object/tesla_wtb_cn.html。
- 想了解面向娱乐的 GeForce 的相关信息,请访问:http://www.nvidia.cn/object/geforce_family_cn.html。
- 想了解面向专业可视化的 Quadro,请访问:http://www.nvidia.cn/object/wtb_workstation_cn.html。

下面,开始 CUDA 的安装过程。

① 确认计算机中使用的是哪款 GPU,是否是 NVIDIA 公司的产品,对于 Windows 平台的用户,可按照以下步骤进行检查:

- 右击桌面,如果在弹出窗口中看到"NVIDIA 控制面板"或"NVIDIA 显示器"等内容,则表示该机中已装有 NVIDIA 的 GPU;
- 在弹出的窗口中单击"NVIDIA 控制面板"或"NVIDIA 显示器",查看"显卡信息",可以看到 GPU 的名称。

通过上述步骤可以得到计算机所使用的 GPU 的型号。

② 确认该 GPU 是否支持 CUDA。访问 http://www.nvidia.cn/object/cuda_gpus_cn.html,查看列表,看该 GPU 是否列在其中。如果列表中有该 GPU,则表示计算机已配有支持 CUDA 的 GPU,可以利用 CUDA 来加速应用程序的性能。

③ 安装 CUDA 产品的驱动程序。在 http://www.nvidia.cn/drivers 中选择对应的操作系统和 GPU 型号,下载相关的 CUDA 驱动程序并按步骤安装。

④ 安装 CUDA Toolkit。目前,NVIDIA 提供的 CUDA Toolkit 支持 Windows(32 位与 64 位版本)和许多不同的 Linux 版本,其最新版本为 3.2,下载地址为 http://developer.nvidia.com/object/cuda_downloads.html。选择相应的操作系统,下载并安装。CUDA Toolkit 需要配合 C/C++编译器。在 Windows 下,目前只支持 Visual Studio 7.x、Visual Studio 8(包括免费的 Visual Studio C++ 2005 Express)及更高版本。Visual Studio 6 和 gcc 在 Windows 下是不支持的,而在 Linux 下则只支持 gcc。

⑤ 安装 GPU Computing SDK code samples。下载地址与上步 CUDA Toolkit 的下载地址相同。该包包括了 NVIDIA 官方提供的程序实例和一些经典算法。

2. 使用设置

下面,分别介绍 Windows 和 Linux 操作系统下 CUDA 的使用设置。

1）Windows 操作系统

在 CUDA Toolkit 安装完成之后，默认会安装在 C:\CUDA 目录中。该目录包含如下几个目录：

- bin——工具程序及动态链接库；
- doc——文件；
- include——头文件；
- lib——链接库档案；
- open64——基于 Open64 的 CUDA 编译器；
- src——一些原代码。

安装程序需要对一些环境变量进行设置，包括：

- CUDA_BIN_PATH——工具程序所在目录，默认为 C:\CUDA\bin；
- CUDA_INC_PATH——头文件文件所在目录，默认为 C:\CUDA\inc；
- CUDA_LIB_PATH——链接库文件所在目录，默认为 C:\CUDA\lib。

下面是在 Visual Studio 环境下如何添加 CUDA 的相关规则：

① 建立一个 Win32 Console 模式的 project（在 Application Settings 中请选 Empty project），并新增一个档案，例如 main. cu。

② 在 main. cu 上单击右键，选择 Properties。选择 General，确定 Tool 的部分是选择 Custom Build Tool。

③ 选择 Custom Build Step，在 Command Line 进行如下设置。

Release 模式：
```
"$(CUDA_BIN_PATH)\nvcc.exe" -ccbin "$(VCInstallDir)bin" -c
-DWIN32 -D_CONSOLE -D_MBCS -Xcompiler
/EHsc,/W3,/nologo,/Wp64,/O2,/Zi,/MT
-I"$(CUDA_INC_PATH)"
-o $(ConfigurationName)\$(InputName).obj $(InputFileName)
```

Debug 模式：
```
"$(CUDA_BIN_PATH)\nvcc.exe" -ccbin "$(VCInstallDir)bin"
-c -D_DEBUG -DWIN32 -D_CONSOLE -D_MBCS
-Xcompiler
/EHsc,/W3,/nologo,/Wp64,/Od,/Zi,/RTC1,/MTd -I"$(CUDA_
INC_PATH)"
-o $(ConfigurationName)\$(InputName).obj $(InputFileName)
```

④ 如果想使用软件模拟 GPU 运行（EMU），可新增两个如下的额外设置。

EmuRelease 模式：
```
"$(CUDA_BIN_PATH)\nvcc.exe" -ccbin "$(VCInstallDir)
bin" -devi ceemu -c -DWIN32 -D_CONSOLE -D_MBCS
-Xcompiler /EHsc,/W3,/nologo,/Wp64,/O2,
```

/Zi,/MT -I"$(CUDA_INC_PATH)"
-o $(ConfigurationName)\$(InputName).obj $(InputFileName)

EmuDebug 模式："$(CUDA_BIN_PATH)\nvcc.exe" -ccbin "$(VCInstallDir)bin" -deviceemu -c -D_DEBUG -DWIN32 -D_CONSOLE -D_MBCS
-Xcompiler /EHsc,/W3,/nologo,/Wp64,/Od,
/Zi,/RTC1,/MTd -I"$(CUDA_INC_PATH)"
-o $(ConfigurationName)\$(InputName).obj $(InputFileName)

⑤ 对所有的配置文件，在 Custom Build Step 的 Outputs 中加入：
$(ConfigurationName)\$(InputName).obj

⑥ 选择 project，右键单击选择 Properties，再选择 Linker。修改所有配置文件的设置，如下：
General/Enable Incremental Linking：No
General/Additional Library Directories：$(CUDA_LIB_PATH)
Input/Additional Dependencies：cudart.lib

2) Linux 操作系统

Linux 环境下 CUDA 的安装步骤与 Windows 下大致相同。因此，这里只叙述其关键步骤，其余步骤参考 Windows 下 CUDA 的安装。

① 安装 CUDA Driver。在终端安装 CUDA 驱动程序，使用 sh 命令执行。如何安装 NVIDIA 的 Linux 驱动程序，请参考 NVIDIA Accelerated Linux Driver Set README and Installation Guide 或访问 http://us.download.nvidia.com/XFree86/Linux-x86/1.0-9755/README/index.html。安装完毕之后，可在 Terminal 中执行[nvidia-xconfig -query-gpu-info]，来查看安装的 NVIDIA GPU。运行结果如图 3-33 所示。

② 安装 CUDA Toolkit。用 sh 命令执行 NVIDIA_CUDA_Toolkit_1.1_*_x86*.run。

安装程序会要求输入安装路径或接受其默认值。推荐以 root 账号身份进行安装并使用默认的安装路径(/usr/local)。之后，将会以<CUDA_INSTALL_PATH>代替实际的安装路径。将 CUDA 的二进制文件(NVCC)和函数路径(libcuda.so)添加到环境变量 PATH 和 LD_LIBRARY_PATH 中。

③ 安装 CUDA SDK。用 sh 命令执行 NVIDIA_CUDA_SDK_1.1_Linux.run。

安装程序会要求输入安装路径或接受其默认值，默认的安装路径为用户的根目录(/NVIDIA_CUDA_SDK)。之后，将会以<SDK_INSTALL_PATH>代替实际的安装路径，在根目录下的.bash_profile 中加入如下几行即可：

```
PATH = $PATH:<CUDA_INSTALL_PATH>/bin
LD_LIBRARY_PATH = $LD_LIBRARY_PATH:<CUDA_INSTALL_PATH>/lib
```

```
export PATH
export LD_LIBRARY_PATH
```

详细步骤可以参考 SDK 安装目录中附带的帮助文档，文档的默认目录为：

～\NVIDIA CUDA SDK\doc\CUDA_SDK_release_notes_linux.txt.

对 CUDA 安装和设置还存在疑问的读者，请参考：

http://blog.csdn.net/OpenHero/archive/2008/11/15/3307166.aspx，该网页是本节重要参考文献之一《GPU 高性能运算之 CUDA》作者赵开勇先生的 CSDN 博客。其中，专门为 CUDA 安装和设置录制了一个名为 CUDA easy start up 的视频教程。该博客还有许多有助于 CUDA 程序员的启蒙文章和对 CUDA 技术的独到见解，还可以给注册会员提供在线答疑。

```
[root@gale45 ~]# nvidia-xconfig -query-gpu-info
Number of GPUs: 4

GPU #0:
  Name       : GeForce GTX 295
  PCI BusID : PCI:3:0:0
  Number of Display Devices: 0

GPU #1:
  Name       : GeForce GTX 295
  PCI BusID : PCI:4:0:0
  Number of Display Devices: 1

  Display Device 0 (CRT-1):
     No EDID information available.

GPU #2:
  Name       : GeForce GTX 295
  PCI BusID : PCI:7:0:0
  Number of Display Devices: 0

GPU #3:
  Name       : GeForce GTX 295
  PCI BusID : PCI:8:0:0
  Number of Display Devices: 0

[root@gale45 ~]# ~
```

图 3-33　Linux 环境下 CUDA Driver 的安装

3. 测试安装

在 Visual Studio 中新建一个项目，将该项目的类型选择为 CUDA（因为之前在 Visual Studio 环境中添加了 CUDA 的相应规则。如果没有该选项，则需要重新检查 CUDA 安装与环境设置的相关步骤），如图 3-34 所示。

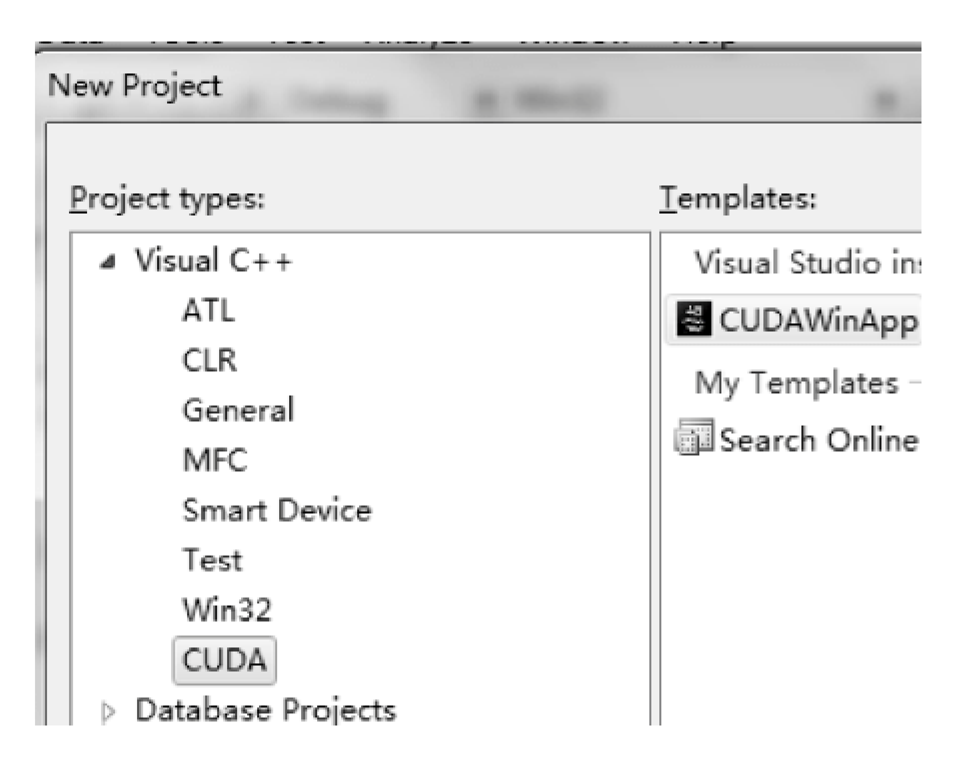


图 3-34　在 Visual Studio 中新建的 CUDA 项目

如果安装了 CUDA Windows Application Wizard，将会看到如图 3-35 所示的界面。



图 3-35　CUDA Visual Studio Wizard

在左侧的 Solution Explorer 中，右键单击新建立的 project，选择 add→new item，再选择 CUDA，这时在 project 中就会形成一个后缀名为.cu 的文件，如图 3-36 所示。

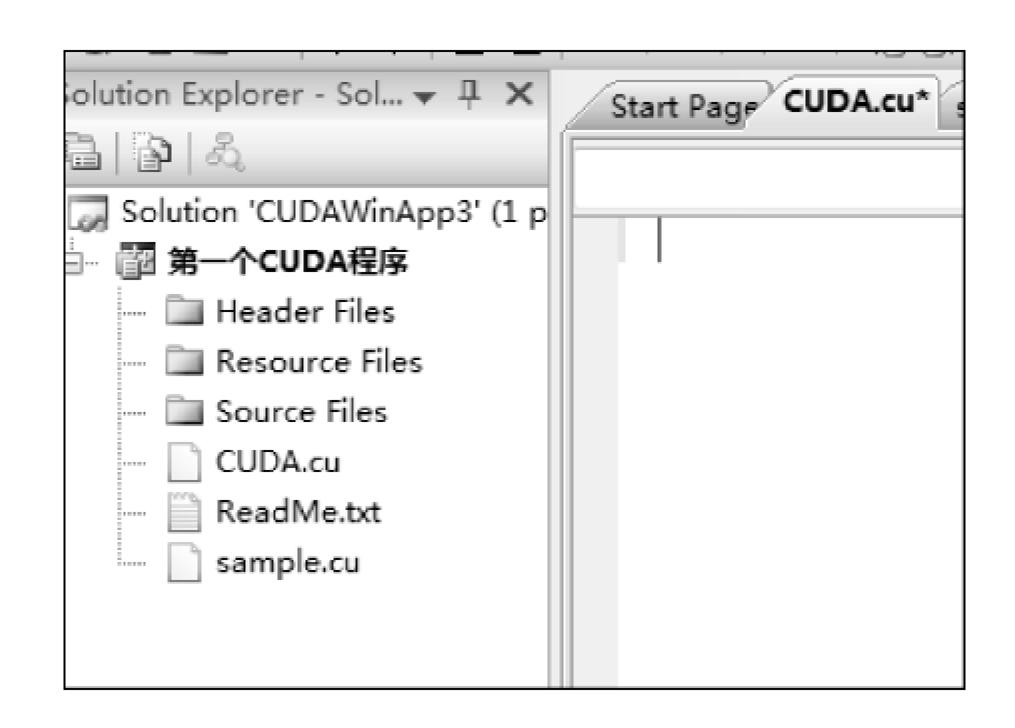


图 3-36 Visual Studio 中的 CUDA 程序

下面，通过运行 SDK 中的程序实例 oceanFFT 测试 CUDA 环境是否安装与设置成功。oceanFFT 使用 FFT 算法模拟海平面。

定位到如下的路径：

```
X:\...\ProgramData\NVIDIA Corporation\NVIDIA GPU ComputingSDK\C\src\oceanFFT\
```

其中，X 为安装 SDK 的盘符。选择使用 Visual Studio 打开文件 oceanFFt. sln。如果 SDK 的版本较低，Visual Studio 会提示将程序的代码进行转换，该转换过程自动进行，无须干涉。此时可以看到该程序的结构（其中，. h 为定义的头文件，. cpp 为封装的 C 语言包，. cu 为包含 kernel 函数的 CUDA 文件），如图 3-37 所示。

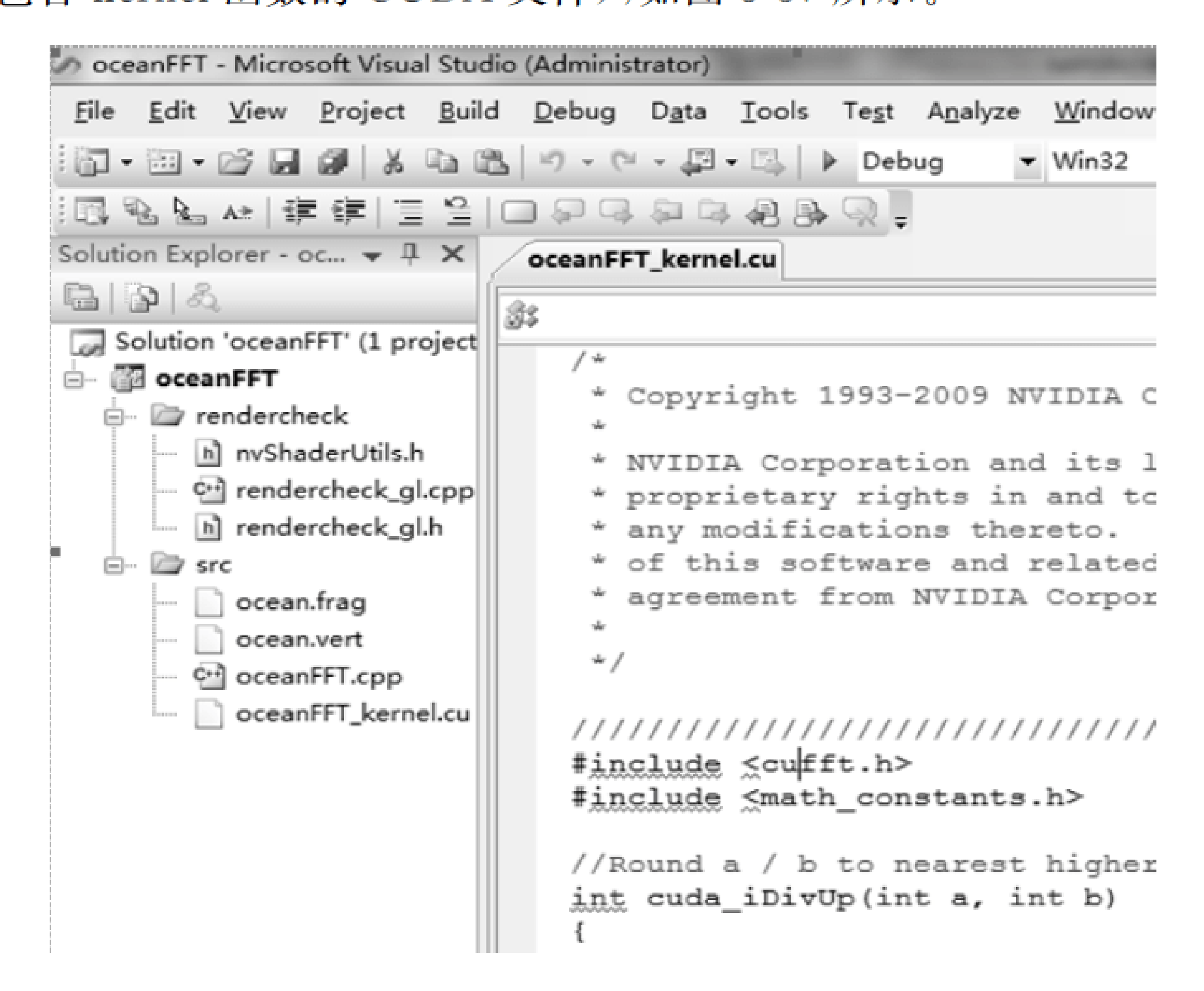


图 3-37 Visual Studio 中程序 oceanFFT 的结构

其中,后缀名为.cu 的文件为 CUDA 程序文件。运行 debug 执行该实例,其执行结果如图 3-38 所示。

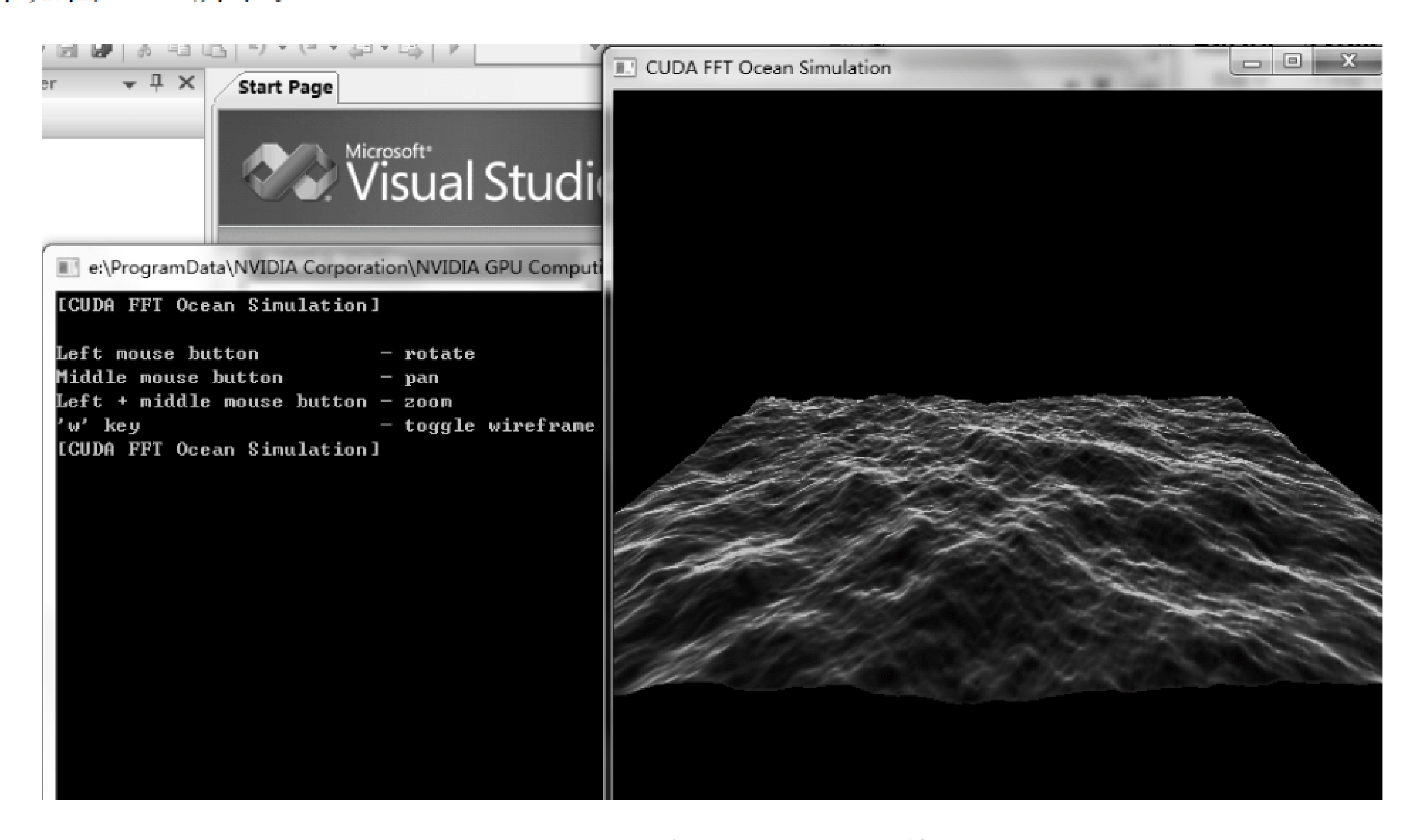


图 3-38　CUDA SDK 中程序 oceanFFT 的执行结果

在图 3-38 中,左侧为执行过程,右侧为动态显示的效果图(对海平面进行模拟)。

下面,在 Linux 环境下执行 SDK 中的 DeviceQuery,获得进行 CUDA 运算的 GPU 的详细信息。

```
创建 SDK project 范例程序
cd <SDK_INSTALL_PATH>
Build:
        - release 输入 "make".
        - debug 输入 "make dbg = 1".
        - emurelease 输入 "make emu = 1".
        - emudebug 输入 "make emu = 1 dbg = 1".
```

在＜SDK_INSTALL_PATH＞执行 make,创建范例程序所使用的公共工具 libcutil。libcutil 是为了方便使用而提供的,不属于 CUDA 的一部分,且撰写的程序也不需要用到它。

然后,执行＜SDK_INSTALL_PATH＞/bin/linux32/release/deviceQuery 进行测试。而 debug、emurelease、emudebug 等其他目录为/bin/linux32/[debug|emurelease|emudebug]。

可以看出,使用 CUDA SDK 易于创建新的 CUDA 程序,复制与修改 CUDA SDK 提

供的项目 template 可满足各种不同的需求。

3.4.3 第一个 CUDA 程序

安装与设置好 CUDA 环境之后，可运行 CUDA Wizard 中自带的一个示例程序——helloCUDA，双击 project 中的 sample.cu 文件。

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <cutil.h>
/* EMU 模式下(以软件模拟方式运行)，因为设备为虚拟，初始化直接返回 ture */
#if _DEVICE_EMULATION_
bool InitCUDA(void){return true;}
#else
 bool InitCUDA(void)                    /* 初始化设备 */
 {
    int count = 0;
    int i = 0;
    cudaGetDeviceCount(&count);         /* 获取设备数目 */
    if(count == 0)
        {
        fprintf(stderr, "There is no device.\n");
        return false;
        }
    for(i = 0; i < count; i++)
        {
        /* 获取支持 CUDA 的设备数目 */
        cudaDeviceProp prop;
        if(cudaGetDeviceProperties(&prop, i) == cudaSuccess)
            {
            if(prop.major >= 1)
              {
                break;
              }
           }
        }
    if(i == count)
   {
        fprintf(stderr, "There is no device supporting CUDA.\n");
        return false;
    }
    cudaSetDevice(i);                               /* 设置要使用的设备 */
    printf("CUDA initialized.\n");
```

```
    return true;
}
#endif
_global_ static void HelloCUDA(char * result, int num)            /* 定义 kernel 函数 */
{
    int i = 0;
    char p_HelloCUDA[] = "Hello CUDA!";
    for(i = 0; i< num; i++)
     {
      result[i] = p_HelloCUDA[i];
     }
}
int main(int argc, char * argv[])                        /* 主程序 */
{
    if(!InitCUDA())
   {
        return 0;
     }
    char *device_result = 0;
    char host_result[12] = {0};
    /* 在显存开辟空间用来存放结果 */
    CUDA_SAFE_CALL(cudaMalloc((void **) &device_result, sizeof(char) * 11));
    unsigned int timer = 0;                              /* 设置定计时器 */
    CUT_SAFE_CALL(cutCreateTimer(&timer));
    CUT_SAFE_CALL(cutStartTimer(timer));
    /* 调用 kernel 函数,在设备上执行 */
    HelloCUDA<<<1, 1, 0>>>(device_result, 11);
    CUT_CHECK_ERROR("Kernel execution failed\n");
    /* 统计时间,在此对所有线程进行同步操作 */
    CUDA_SAFE_CALL(cudaThreadSynchronize());
    CUT_SAFE_CALL(cutStopTimer(timer));
    printf("Processing time: %f (ms)\n", cutGetTimerValue(timer));
    CUT_SAFE_CALL(cutDeleteTimer( timer));
    CUDA_SAFE_CALL(cudaMemcpy(&host_result, device_result, sizeof(char) * 11,
    cudaMemcpyDeviceToHost));                  /* 将计算产生的结果拷贝至主机端主存 */
    printf("%s\n", host_result);
    CUDA_SAFE_CALL(cudaFree(device_result));
    CUT_EXIT(argc, argv);
    return 0;
}
```

对该 CUDA 程序进行编译、连接并运行，其运行结果如下所示：

```
CUDA initialized.
Processing time: 1.743265 (ms)
Hello CUDA!

Press ENTER to exit...
```

在上面的 CUDA 程序中，黑体部分为与 CUDA 相关的 runtime API。kernel 函数为设备端运行的代码部分，其余部分为主机端运行的代码部分。

该 CUDA 程序的主要流程为：

- CUDA 初始化　其中包括获取设备信息、选定所使用设备；
- 定义 Kernel 函数；
- 在设备端建立数据；
- 设置计时器；
- 在设备端调用 Kernel 函数；
- 将结果返回主机端；
- 显示结果。

事实上，大多数情况下，计算所需的数据在主机端生成。此时，需要首先将计算所需数据拷贝到设备上，再调用 Kernel 函数。

3.4.4　CUDA 编译器

NVCC 编译器根据配置编译 CUDA C 代码，可以生成三种不同的输出：PTX、CUDA 二进制序列和标准 C。NVCC 是一种编译器驱动。通过命令行选项，NVCC 可以在编译的不同阶段启动不同的工具完成编译工作。

NVCC 工作的大致流程是：首先将代码划分为 Host 端和 Device 端部分。其中，Host 端部分由 C/C++高级编译器编译；Device 端部分由 NVCC 编译成 PTX 中间代码或 cubin 对象，当用户使用 CUDA 驱动 API 时，PTX 中间代码或 cubin 对象就可直接被设备执行。其过程如图 3-39 所示。

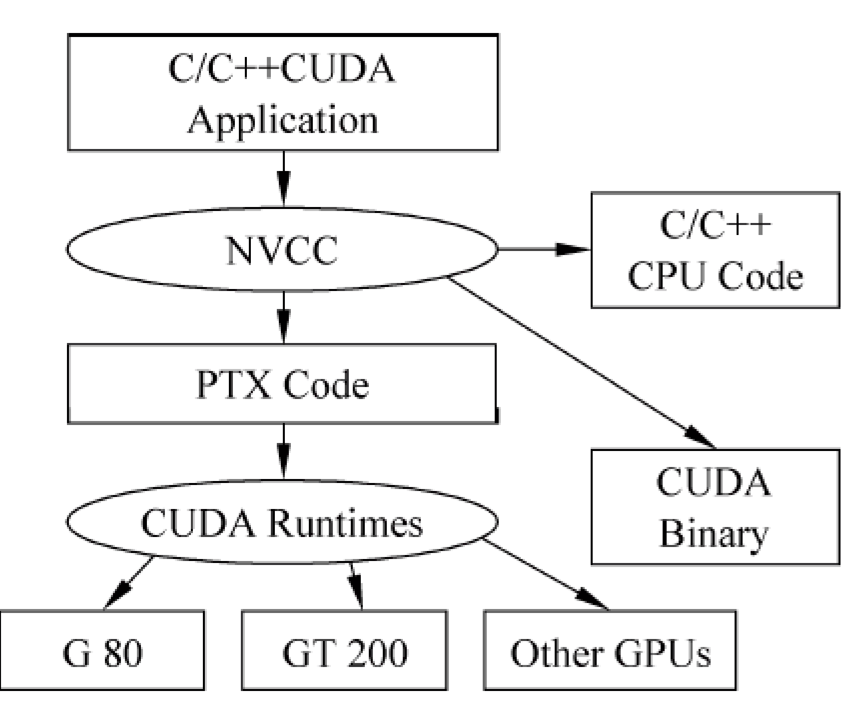


图 3-39　NVCC 编译器的流程示意图

NVCC 编译器的前端按照 C++的语法规则处理 CUDA 源文件。主机代码完全支持 C++。但是，设备代码只支持 C++的 C 子集，C++的一些特定功能（比如类、继承等）不被支持。由于使用了 C++语法规则的原因，空指针（比如 malloc()函数调用返回的值）未经过类型的强制转换，因此不能赋值给非空指针。

NVCC 引入了两个编译指令：noinling 与＃pragma unroll。其中，

- __ noinline __指令表示在默认的情况下__ device __函数始终为内联函数。但是，__ noinline __函数限定符可用于编译器提示以尽量不内联函数。函数体必须位于调用该函数的同一文件中。对于带有指针参数的函数和具有大型参数列表的函数，编译器将忽略__ noinline __限定符。

- ＃pragma unroll 指令可用于控制任何给定循环的展开。它必须放置在循环之前，并只应用于该循环。它后面可以跟一个数字，用来指定循环必须展开的次数。

3.4.5　CUDA 常用 API

关于 CUDA 常用 API(编程接口)的详细信息可以参考 NVIDIA CUDA 的用户手册。

使用 CUDA API 可使熟悉 C 编程语言的程序员很容易地编写在设备上执行的程序。它包括 C 语言的最小扩展集合，允许程序员确定需要在设备上执行的部分源代码。

CUDA 的运行时库可被划分为：

- 主机组件　在主机上运行，提供函数以控制并访问主机中一个或多个计算设备；
- 设备组件　在设备上运行，提供特定于设备的函数；
- 通用组件　提供内置的向量类型和在主机与设备代码中都支持的 C 标准库子集。

只有那些支持在设备上运行的 C 标准库中的函数才是公共运行时(runtime)提供的函数。

C 编程语言的扩展有以下几部分。

1. 函数类型限定符

用于指定函数是在主机上还是在设备上执行或从主机还是设备中调用。主要有以下三种限定符。

① __device__限定符：申明函数在设备上执行，只能从设备中调用。

② __global__限定符：将函数声明为内核函数，在设备上执行，只能从主机中调用。

③ __host__限定符：声明函数在主机上执行，只能从主机中调用。它等同于仅使用__host__声明的函数或不使用__host__、__device__或__global__限定符的任意一个声明函数。不管是哪一种情况，该函数仅为主机编译。__host__限定符还可以与__device__限定符结合使用，此时，该函数同时为主机和设备编译。

其中，__device__和__global__函数不支持迭代，不能在函数体内声明静态变量，参数不可变；__device__函数不能取其地址，__global__函数的函数指针则被支持；__global__和__host__限定符不能一起使用；__global__函数必须具有 void 返回类型；对__global__函数的任何调用必须指定其执行设置；__global__函数的参数通过共享内存传递给设备并被限定为 256 字节。

2. 变量类型限定符

用于指定变量在设备上的内存位置。主要有以下三种限定符。

① __device__限定符：声明驻留在设备上的变量。

在其他类型的变量限定符中，至多有一个可与__device__一起使用，以进一步指定变

量属于哪个内存空间。如果其中任何一个都不出现，则该变量驻留在全局内存空间中，具有应用程序的生命期，可通过运行时库从 grid 的所有线程中和从主机中访问。

② __constant__限定符：可与__device__一起使用，声明变量。__constant__限定符指定驻留在常量内存空间中的变量，具有应用程序的生命期，可通过运行时库从 grid 的所有线程中和从主机中访问。

③ __shared__限定符：可与__device__一起使用，声明变量。__shared__限定符指定驻留在线程块的共享内存空间中的变量，具有 bolck 的生命期，仅可从 block 内所有的线程中访问。

以__shared__方式声名的线程中的共享变量具有完全顺序一致性，但是在线程中 kernel 并不一定会按照统一的线性顺序执行。仅在执行同步语句__syncthreads()之后，来自其他线程的写入才能保证可见。

数组的大小在启动时被确定。以上述三种方式声明的所有变量在内存中从同一地址开始，故数组中的变量必须通过偏移量进行管理。这些限定符不允许用于 struct 和 union 成员、形参以及在主机上执行的函数内部的本地变量。

其中，__shared__和__constant__变量已经隐含了静态存储；__device__、__shared__和__constant__变量不能使用 extern 关键字定义为外部变量；__device__和__constant__变量仅允许用于文件范围；__constant__变量不能从设备中赋值，只能从主机中通过主机运行时函数来赋值。

3. 执行设置

用于指定如何从主机中的设备上执行内核。

对__global__类型函数的任何调用必须为该调用指定执行设置。执行设置定义将用于在设备上执行函数的 grid 和 block 的维度以及相关联的流。通过在函数名称和圆括号括起的参数列表之间插入<<< Dg, Db, Ns, S >>>形式的表达式，就可定义执行设置，其中：

- Dg 是类型 dim3，用于指定 grid 的维度和大小。因此，Dg. x * Dg. y 等于要启动的 block 数；Dg. z 未使用。
- Db 是类型 dim3，用于指定每个 block 的维度和大小。因此，Dg. x * Dg. y * Db. z 等于每个 block 的线程数。
- Ns 是类型 size_t，用于指定为该调用按 block 动态分配的共享内存中的字节数以及静态分配的内存。此动态分配的内存由声明为外部数组的任何一个变量使用。Ns 是默认值为 0 的可选参数。
- S 是类型 cudaStream_t，用于指定相关的流。S 是默认值为 0 的可选参数。

4. 四个内置变量

用于指定 grid 和 block 的维度，以及 block 和线程索引。

- gridDim：此变量的类型为 dim3，包含 grid 的维度。
- blockIdx：此变量的类型为 uint3，包含 grid 中的 block 索引。
- blockDim：此变量的类型为 dim3，包含 block 的维度。
- threadId：此变量的类型为 uint3，包含 block 中的线程索引。

3.4.6　CUDA 编程模型

CUDA 编程模型将 CPU 作为主机，GPU 作为协处理器。在一个系统中，可以存在一个主机和若干个设备。GPU 负责进行逻辑性强的事务处理和串行计算，GPU 则负责执行高度线程化的并行任务。CPU 与 GPU 各自拥有相互独立的存储器地址空间，CPU 拥有主机端的内存，GPU 拥有设备端的显存。CUDA 对内存的操作与一般的 C 程序基本相同，但是增加了一种新的 pinned memory。显存则需要调用 CUDA API 中的存储器管理函数来操作，这些管理操作包括开辟、释放和初始化显存空间，以及在主机端和设备端之间进行数据传输等。

内核函数(kernel)以线程网格(grid)的形式进行组织，每个 grid 由若干个线程块(block)组成，而每个线程块又由若干个线程组成。实际上，kernel 是以 block 为单位进行执行的。各个 block 并行执行，之间无法通信，也没有执行顺序，而 thread 则可以通过共享内存(shared memory)进行通信。图 3-40 是 CUDA 的编程模型示意图。

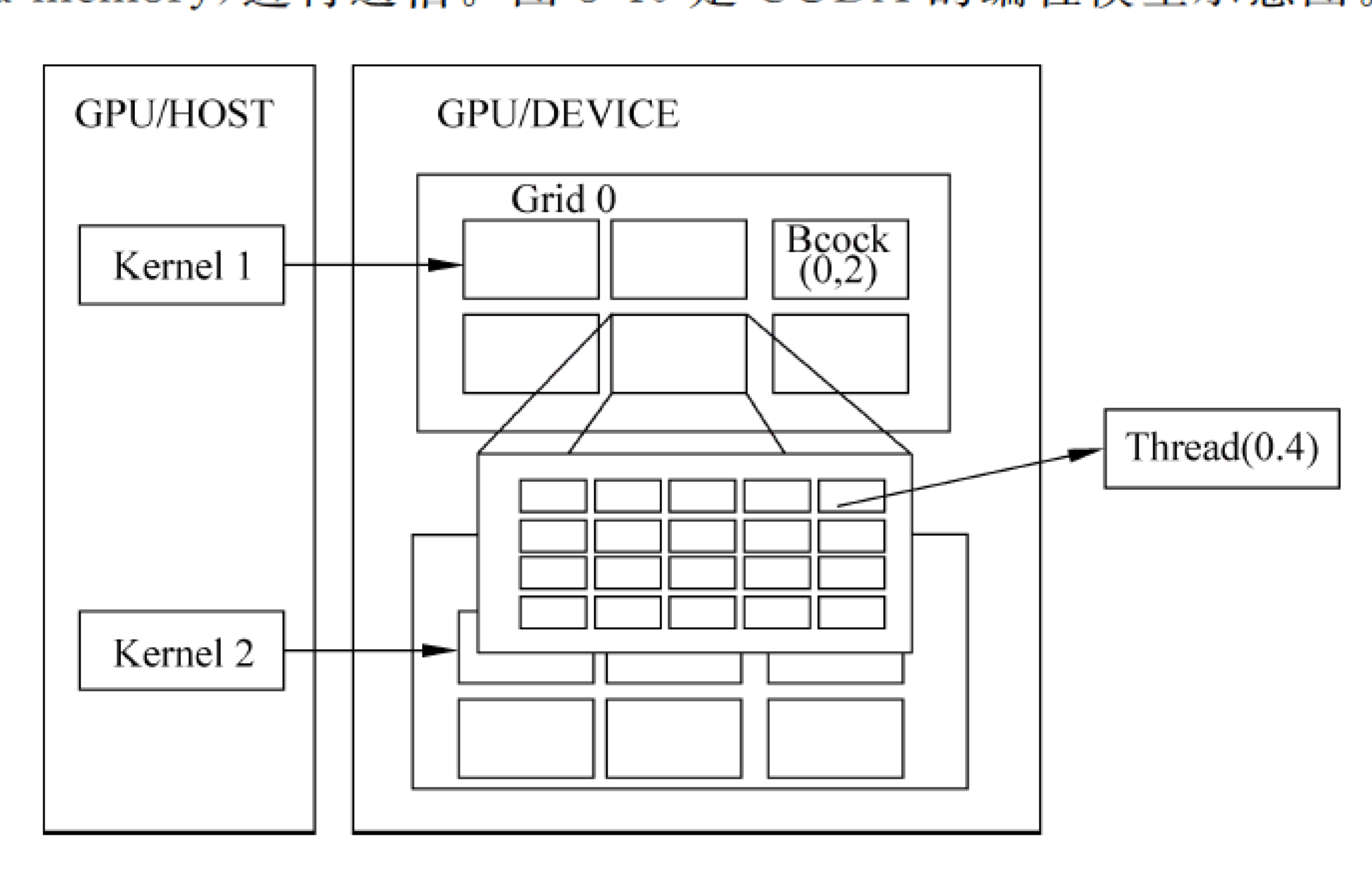


图 3-40　CUDA 的编程模型示意图

GPU 不能直接存取主存，只能存取显存。因此，需要将数据从主存先复制到显存中，进行运算之后，再将运算的结果从显存复制到主存中。这些复制动作受限于 PCI

Express 的速度。使用 PCI Express x16 时，PCI Express 1.0 可以提供双向各 4GB/s 的带宽，而 PCI Express 2.0 则可提供双向各 8GB/s 的带宽。

从一般的内存复制数据到显存的时候，由于一般的内存可能随时会被操作系统移动，因此 CUDA 会先将数据复制到一块内部的内存之后，才会利用 DMA 将数据复制到显存中。如果想避免这个重复的复制动作，可以使用 cudaMallocHost 方式，在主存中取得一块页锁定的内存。不过，如果要求的页锁定内存的量太大，将会影响到操作系统对内存的管理，可能会减低系统的效率。

一旦确定了程序中的并行部分，就可以考虑把这部分计算工作交给 GPU，称为 kernel 函数。kernel 函数必须通过__ global __函数类型限定符进行定义，并且只能在主机端代码中被调用。一个 kernel 函数并不是一个完整的程序，而是整个 CUDA 程序中可以被并行执行的部分。将按照程序中相应语句的顺序执行这些部分，从而满足顺序一致性。

在执行 CUDA 程序的时候，每个 stream processor(流处理器)对应一个 thread，每个 multiprocessor(多处理器)则对应一个 block，每个 multiprocessor 有 8 个 stream processor。在执行程序过程中，CUDA 以 warp 为单位，一个 warp 中有 32 个 thread，被分成两组 half-warp(各有 16 个 thread)。由于 multiprocessor 中并没有太多别的存储器内存，因此每个 thread 的状态都是直接保存在 multiprocessor 的寄存器中。所以，如果在一个 multiprocessor 中同时执行的 thread 数越多，则需要越多的寄存器空间。

3.4.7 CUDA 存储器模型

CUDA 规定了多级的存储模型，使得 CUDA 程序中线程的运行更加高效、灵活。每个 thread 拥有私有存储器、寄存器和局部存储器，每个 block 拥有一个共享存储器。除此之外，还有两种可被所有线程访问的只读存储器：常数存储器(constant memory)和纹理存储器(texture memory)。它们分别为不同的应用进行了优化。另外，pinned memory 是在主机端主存中开辟的一部分页锁定内存。表 3-11 给出了各种存储器所在位置、缓存情况与访问权限。

其中，所有的 SM(Stream Multiprocessor)共享全局存储器，每个 SM 拥有自己的共享存储器。指令单元每发射一条指令，片上的 8 个流处理器就同时执行这条指令，而每个流处理器拥有一块寄存器。图 3-41 为 CUDA 存储器模型的组织结构图。

表 3-11 各级存储器比较

存储器	位置	是否拥有缓存	访问权限
register	片内	N/A	device 可读写
local memory	板载显存	无	device 可读写
shared memory	片内	N/A	device 可读写

续表

存　储　器	位　　置	是否拥有缓存	访 问 权 限
constant memory	板载显存	有	device 可读，host 可读写
textrue memory	板载显存	有	device 可读，host 可读写
global memory	板载显存	无	device 可读，host 可读写
host memory	Host 内存	无	host 可读写
pinned memory	Host 内存	无	host 可读写

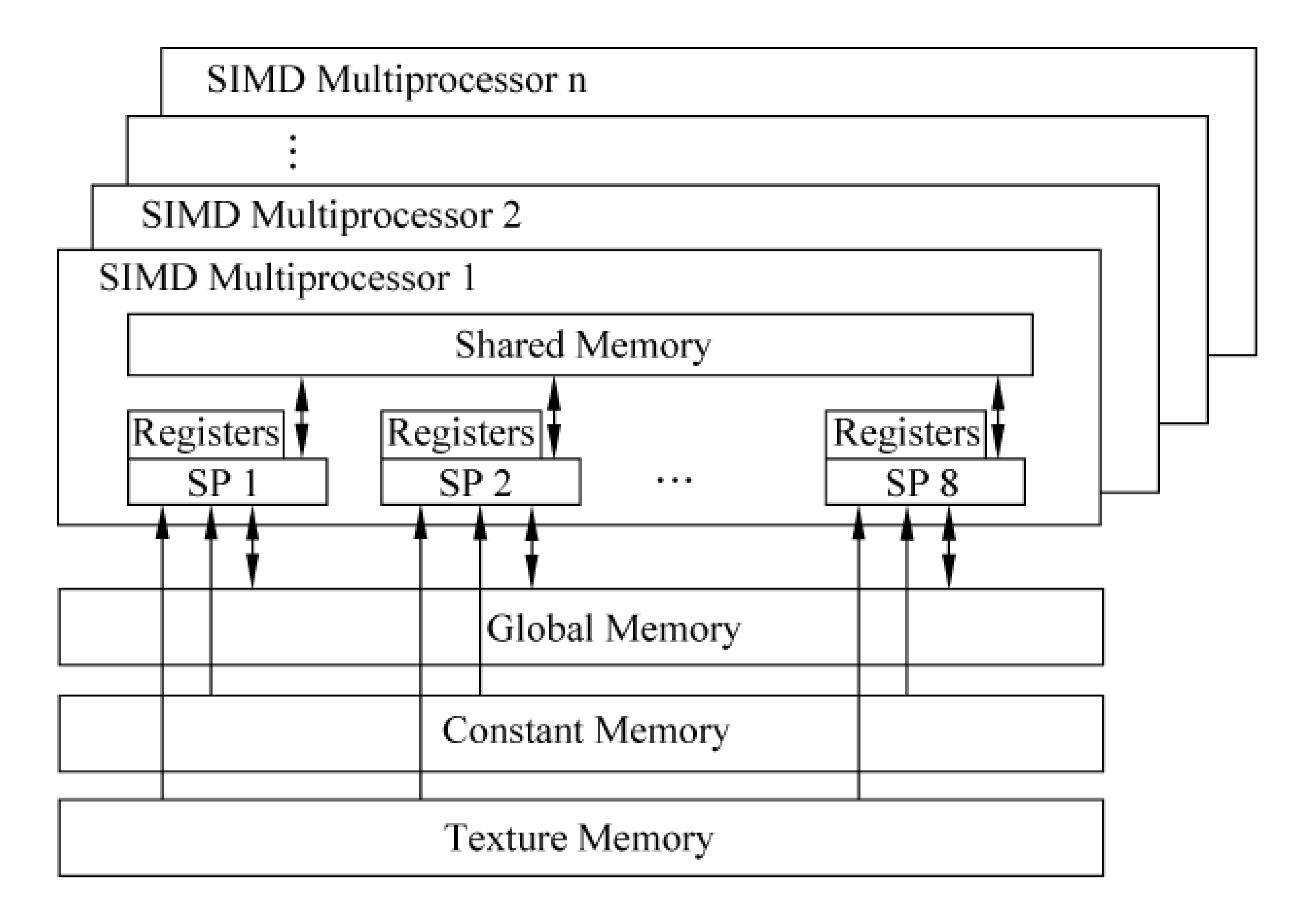


图 3-41　CUDA 存储器模型的组织结构图

3.4.8　编程实例的运行、分析与优化

在了解 CUDA 基础知识之后，本节将使用一个实际的程序进行编程实践。

前面介绍的 Hello CUDA！是 Visual Studio Wizard 自带的一个示例程序，是 CUDA 的入门程序，并无多大实用价值。因此，在这里给出一个有实用价值的程序。为了与后续章节 Cell BE 部分的实例程序相一致，同时也为了介绍 CUDA 相关的一些性质，在这里选用的实例程序为矩阵乘法。该实例参考自《深入浅出谈 CUDA》一文，有兴趣的读者可以通读此文的其他部分以获取更深入的了解。

为简单起见，这里以方阵为例。假设，有两个矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$，使用如下代码计算 $\boldsymbol{A}\times\boldsymbol{B}=\boldsymbol{C}$：

```
for(i = 0; i < n; i++)
{
    for(j = 0; j < n; j++)
    {
        C[i][j] = 0;
```

```
        for(k = 0; k < n; k++)
        {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

先准备好产生数据、设定 CUDA 等工作。主函数代码如下：

```
int main()
{
    float * a, * b, * c, * d;
    int n = 1000;
    if(! InitCUDA())
    return 0;
    a = (float * ) malloc(sizeof(float) * n * n);
    b = (float * ) malloc(sizeof(float) * n * n);
    c = (float * ) malloc(sizeof(float) * n * n);
    d = (float * ) malloc(sizeof(float) * n * n);
    srand(0);
    matgen(a, n, n);
    matgen(b, n, n);
    clock_t time = matmultCUDA(a, n, b, n, c, n, n);
    matmult(a, n, b, n, d, n, n);
    compare_mat(c, n, d, n, n);
    double sec = (double) time / CLOCKS_PER_SEC;
    printf("Time used: %.2f (%.2lf GFLOPS)\n", sec, 2.0 * n * n * n / (sec * 1E9));
    return 0;
}
/* CUDA 的初始化函数 InitCUDA()  */
bool InitCUDA()
{
    int count;
    cudaGetDeviceCount(&count);
    if(count == 0)
    {
        fprintf(stderr, "There is no device.\n");
        return false;
    }
    int i;
    for(i = 0; i < count; i++)
    {
        cudaDeviceProp prop;
        if(cudaGetDeviceProperties(&prop, i) == cudaSuccess)
        {
```

```
                if(prop.major >= 1)
                {
                    break;
                }
            }
        }
        if(i == count)
        {
            fprintf(stderr, "There is no device supporting CUDA 1.x.\n");
            return false;
        }
        cudaSetDevice(i);
        return true;
}
/* 产生矩阵的函数 matgen   */
void matgen(float * a, int lda, int n)
{
        int i, j;
        for(i = 0; i < n; i++)
        {
            for(j = 0; j < n; j++)
            {
                a[i * lda + j] = (float) rand() / RAND_MAX + (float) rand() / (RAND_MAX * RAND_MAX);
            }
        }
}
/* 函数 matmult 利用随机数生成器将矩阵全填为 0 ~ 1 之间的数字
 * 由于在 C 语言中无法声明可变大小的二维矩阵,所以这里使用 i × lda + j 的方式
 * 进行矩阵乘法   */
void matmult(const float * a, int lda, const float * b, int ldb, float * c, int ldc, int n)
{
        int i, j, k;
        for(i = 0; i < n; i++)
        {
            for(j = 0; j < n; j++)
            {
                double t = 0;
                for(k = 0; k < n; k++)
                {
                    t += a[i * lda + k] * b[k * ldb + j];
                    c[i * ldc + j] = t;
                }
            }
        }
```

```
}
/* 函数 compare_mat 在 CPU 上进行矩阵乘法,用于验证答案正确与否
 * 它使用双精度来存储中间计算结果,以提高精确度.
 * 计算两个矩阵的最大相对误差和平均相对误差,并将结果印出来   */
void compare_mat(const float * a, int lda, const float * b, int ldb, int n)
{
     float max_err = 0;
     float average_err = 0;
     int i, j;
     for(i = 0; i<n; i++)
     {
         for(j = 0; j<n; j++)
         {
             if(b[i * ldb + j] != 0)
             {
                 float err = fabs((a[i * lda + j] - b[i * ldb + j]) / b[i * ldb + j]);
                 if(max_err < err) max_err = err; average_err += err;
             }
         }
     }
    printf("Max error: %g Average error: %g\n",max_err, average_err / (n * n));
}

/* CUDA 的矩阵乘法的部分: 函数 matmultCUDA
 * 该函数在显存中设置存放矩阵的内存,然后把主存中的矩阵数据复制到显存中
 * 这里,使用一个新的 cudaMemcpy2D 函数,用来复制二维数组,可以指定数组的 pitch
 * 这样,调用一次函数就可完成矩阵数据从主存到显存的复制   */
#define NUM_THREADS 256
clock_t matmultCUDA(const float * a, int lda, const float * b, int ldb, float * c, int
ldc, int n)
{
    float *ac, *bc, *cc;
    clock_t start, end;
    start = clock();
    cudaMalloc((void**) &ac, sizeof(float) * n * n);
    cudaMalloc((void**) &bc, sizeof(float) * n * n);
    cudaMalloc((void**) &cc, sizeof(float) * n * n);
    cudaMemcpy2D(ac, sizeof(float) * n, a, sizeof(float) * lda, sizeof(float) * n, n,
    cudaMemcpyHostToDevice);
    cudaMemcpy2D(bc, sizeof(float) * n, b, sizeof(float) * ldb, sizeof(float) * n, n,
    cudaMemcpyHostToDevice);
    int blocks = (n + NUM_THREADS - 1) / NUM_THREADS; matMultCUDA<<<blocks * n, NUM_THREADS
    >>> (ac, n, bc, n, cc, n, n); cudaMemcpy2D(c, sizeof(float) * ldc, cc, sizeof(float) *
    n, sizeof(float) * n, n,
```

```
    cudaMemcpyDeviceToHost);
    cudaFree(ac);
    cudaFree(bc);
    cudaFree(cc);
    end = clock();
    return end - start;
}

/* 函数 matMultCUDA 是完成计算的 kernel 函数
 * 由 bid 和 tid 计算出该 thread 需要计算的 row 和 column,在判断 row 和 column
 * 在范围内之后,进行矩阵乘法计算,并将计算结果写到矩阵 c 中   */
__global__ static void matMultCUDA(const float * a, size_t lda, const float * b, size_t
ldb, float * c, size_t ldc, int n)
{
    const int tid = threadIdx.x;
    const int bid = blockIdx.x;
    const int idx = bid * blockDim.x + tid;
    const int row = idx / n; const int column = idx % n;
    int i;
    if(row < n && column < n)
    {
        float t = 0;
        for(i = 0; i<n; i++)
        {
            t += a[row * lda + i] * b[i * ldb + column];
        }
        c[row * ldc + column] = t;
    }
}
```

在 GeForce 8800 GT 上,该程序的实际执行结果如下所示:

```
CUDA initialized.
Max error: 2.01484e-006 Average error: 3.36637e-007 Time used: 1.1560
Press ENTER to exit...
```

从实际执行结果中可以看出两个明显的问题:

- 执行效率相当低;
- 最大相对误差偏高。

计算结果的相对误差偏高的原因是:在 CPU 上进行计算时,使用了双精度(即 64 位浮点数),而在 GPU 上则只能使用 float(32 位浮点数)。在对大量数据进行累加的过程中,由于累加结果可能迅速变大,因此后面的数据很容易被舍去过多的位数。由于,在进行加、减、乘时,CUDA 的浮点数运算符合 IEEE 754 规定的精度,因此,可以利用 Kahan's Summation Formula 提高计算的精度。为此,将相应代码修改为:

```
if(row < n && column < n)
{
    float t = 0; float y = 0;
    for(i = 0; i<n; i++)
    {
        float r; y -= a[row * lda + i] * b[i * ldb + column];
        r = t - y; y = (r - t) + y; t = r;
    }
}
```

执行相应修改之后的程序，其结果如下所示：

```
CUDA initialized.
Max error: 1.19209e-007 Average error: 4.22751e-008 Time used: 1.1670
Press ENTER to exit...
```

从该计算结果来看，计算结果的相对误差已有了很大的改善，但执行效率并无变化。由于 Kahan's Summation Formula 的计算量变大，但是执行效率却有所降低，由此，可以看出该 kernel 函数的主要瓶颈在于内存的存取，因为大量的内存读取是重复的。

例如，a 的一个 row 在每次进行计算时都被重复读入，这是相当浪费的。采用这样的计算方式，总共需要读取 $2\times n^3$ 次内存。如果让一个 row 只需要读入一次的话，则可将内存读取次数减少到为 n^3+n^2 次。

可以利用共享内存来存储每个 row 的数据。不过，因为只有同一个 block 的 thread 可以共享 shared memory，因此一个 row 只能由同一个 block 的 thread 进行计算。另外，也需要能存放整个 row 的 shared memory。

因此，可将调用 kernel 函数的部分修改为：

```
matMultCUDA<<<n, NUM_THREADS, sizeof(float) * n>>> (ac, n, bc, n, cc, n, n);
kernel 函数的部分则被修改为：
__global__ static void matMultCUDA(const float * a, size_t lda, const float * b, size_t
ldb, float * c, size_t ldc, int n)
{
    extern __shared__ float data[];
    const int tid = threadIdx.x;
    const int row = blockIdx.x;
    int i, j;
    for(i = tid; i<n; i += blockDim.x)
    {
    data[i] = a[row * lda + i];
     }
    __syncthreads();
    for(j = tid; j<n; j += blockDim.x)
    {
```

```
            float t = 0;
            float y = 0;
            for(i = 0; i < n; i++)
            {
                float r; y - = data[i] * b[i * ldb + j];
                r = t - y; y = (r - t) + y; t = r;
            }
        c[row * ldc + j] = t;
        }
    }
```

首先把整个 row 读入 shared memory 中，之后进行计算。一个 row 只由一个 block 计算。其执行结果如下所示：

```
CUDA initialized.
Max error: 1.19209e-007 Average error: 4.22751e-008 Time used: 0.4220
Press ENTER to exit...
```

可以看出，计算的结果没有改变，但程序的执行速度却提高了一倍以上。然而，与 GeForce 8800 GT 在理论上拥有超过 300GFLOPS 的运算性能相比，执行效率仍不尽理想(即使把 Kahan's Summation Formula 所需的额外运算考虑进去，这样的执行效率连理论最大值的十分之一都不到)，其原因是对内存的存取次数仍然太多了。虽然现在 a 的 row 数据已不再需要重复读取，但是矩阵 b 的 column 数据仍然被重复读取。

存在的另一问题并不明显：对矩阵 B 的读取，虽然看起来不连续但实际上它是连续的。这是因为不同的 thread 会读取不同的 column，因此将每个 thread 同时读取的各个 column 加起来，就是一个连续的内存块。那为什么执行效率仍然不佳呢？这是因为，GPU 内存控制器从某个固定的倍数地址(例如 16 的倍数)开始读取数据才会获得最高的效率。由于矩阵大小并不是 16 的倍数(在该实例中，使用的是 1000×1000 的方矩)，所以导致程序的执行效率不佳。

为了解决上述问题，可在 cudaMalloc 时进行稍微的修改，让宽度变成适当的倍数即可。然而，到底多少是适当的倍数呢？CUDA 提供的函数 cudaMallocPitch 能够自动以最佳的倍数配置内存。因此，可将该实例程序中的 cudaMalloc 部分修改为：

```
size_t pitch_a, pitch_b, pitch_c;
cudaMallocPitch((void**) &ac, &pitch_a, sizeof(float) * n, n);
cudaMallocPitch((void**) &bc, &pitch_b, sizeof(float) * n, n);
cudaMallocPitch((void**) &cc, &pitch_c, sizeof(float) * n, n);
```

函数 cudaMallocPitch 将以适当的倍数配置内存，并把配置的宽度传回。该宽度将在矩阵被复制到显存时使用：

```
cudaMemcpy2D(ac , pitch_a, a, sizeof(float) * lda, sizeof(float) * n,
```

```
                n,cudaMemcpyHostToDevice);
cudaMemcpy2D(bc , pitch_b, b, sizeof(float) * ldb, sizeof(float) * n,
                n,cudaMemcpyHostToDevice);
```

调用 kernel 函数的部分也需要进行相应的修改：

```
matMultCUDA<<<n, NUM_THREADS, sizeof(float) * n>>> (ac, pitch_a / sizeof(float), bc,
    pitch_b / sizeof(float), cc, pitch_c / sizeof(float), n);
```

同样地，将计算结果复制回主存时，也要使用传回的宽度值：

```
cudaMemcpy2D(c, sizeof(float) * ldc, cc, pitch_c, sizeof(float) * n, n,
    cudaMemcpyDeviceToHost);
```

经过上述修改之后，实例程序的执行结果如下所示：

```
CUDA initialized.
Max error: 1.19209e-007 Average error: 4.22751e-008 Time used: 0.1250
Press ENTER to exit...
```

可以看出，实例程序的执行速度又提高了三倍以上，而这仅需要稍稍修改内存的配置方式。虽然执行速度得到进一步提高，但是与前面提到的理论值仍有相当的差距。这是因为仍需要 n^3+n^2 次内存读取和 n^2 次内存写入。由于 $n=1000$，该实例程序的每个矩阵元素的大小是 32 位，所以总共的内存存取数据量约为 4GB，除以实际执行的时间 0.125s，得到的带宽约为 32GB/s，这已比较接近 GeForce 8800GT 显存的带宽。由于该实例程序在计算花费时间的时候，把配置内存以及数据复制等也计算在内，因此实际上花费在 kernel 函数上的时间应更短。因此，可以看出，该实例程序的执行效率受限于内存带宽。

为了进一步提高该实例程序的执行效率，需进一步降低内存带宽的使用。虽然，矩阵 $\boldsymbol{A}$ 的存取次数已降至最低，但矩阵 $\boldsymbol{B}$ 的存取次数并没有减少。这是因为只将矩阵 $\boldsymbol{A}$ 的 row 数据加载到了 shared memory 中，但矩阵 $\boldsymbol{B}$ 的 column 却还是被重复使用，应该避免矩阵 $\boldsymbol{B}$ 的 column 数据的重复加载。不过，由于矩阵 $\boldsymbol{B}$ 的 column 的使用时机与矩阵 A 的 row 的使用时机不同，所以并不能直接这样做。

其解决方法是将整个矩阵乘法分割为很多小矩阵的乘法。由于，目前 CUDA 每个 block 的 thread 数目最多为 512，因此 $k=16$（k 是分隔出的 block 的规模）似乎是一个相当理想的数字（共 256 个 thread）。因此，对一个 $n=1000$ 的方矩来说，可将内存存取量减少到约 500MB，也就是上面存取量的 1/8（即 500MB/4GB）。理论上，这样做可使效率提高八倍。

让每个 block 有 16×16 个 thread，再建立（n/16）×（n/16）个 block。将调用 kernel 函数的部分修改为：

```
int bx = (n + BLOCK_SIZE - 1) / BLOCK_SIZE; dim3 blocks(bx, bx);
dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
matMultCUDA <<< blocks, threads >>>(ac, pitch_a / sizeof(float), bc, pitch_b / sizeof
(float), cc, pitch_c / sizeof(float), n);
```

其中，BLOCK_SIZE 被定义为 16。dim3 是 CUDA 的一种数据形态，表示它是一个 3D 向量。这里，通过 dim3 建立 16×16 个 thread 的 block 和(n/16)×(n/16)个 block。

Kernel 函数的部分被修改为：

```
__global__ static void matMultCUDA(const float * a, size_t lda, const float * b, size_t
ldb, float * c, size_t ldc, int n)
{
    __shared__ float matA[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float matB[BLOCK_SIZE][BLOCK_SIZE];
    const int tidc = threadIdx.x;
    const int tidr = threadIdx.y;
    const int bidc = blockIdx.x * BLOCK_SIZE;
    const int bidr = blockIdx.y * BLOCK_SIZE;
    int i, j;
    float results = 0;
    float comp = 0;
    for(j = 0; j < n; j += BLOCK_SIZE)
    {
        if(tidr + bidr < n && tidc + j < n)
        {
            matA[tidr][tidc] = a[(tidr + bidr) * lda + tidc + j];
        }
        else
        {
            matA[tidr][tidc] = 0;
        }
        if(tidr + j < n && tidc + bidc < n)
        {
        matB[tidr][tidc] = b[(tidr + j) * ldb + tidc + bidc];
        }
        else
        {
            matB[tidr][tidc] = 0;
        }
        __syncthreads();
        for(i = 0; i < BLOCK_SIZE; i++)
        {
            float t;
```

```
                comp -= matA[tidr][i] * matB[i][tidc];
                t = results - comp;
                comp = (t - results) + comp; results = t;
            }
            __syncthreads();
        }
        if(tidr + bidr < n && tidc + bidc < n)
        {
            c[(tidr + bidr) * ldc + tidc + bidc] = results;
        }
    }
```

由于使用了 16×16 的 thread，因此 threadIdx. x 和 threadIdx. y 的范围分别为 0～15。blockIdx. x 和 blockIdx. y 的范围分别为 0～n/16。在该实例程序中，由于方阵的大小不一定是 16 的倍数，因此需要使用 if 判断语句来检查是否超出了方阵的范围。

执行修改的实例程序，其结果如下所示：

```
CUDA initialized.
Max error: 1.19209e-007 Average error: 4.22751e-008 Time used: 0.0780
Press ENTER to exit...
```

可以看出，实例程序的执行速度虽然提高了，但似乎还未达到预期的八倍。当然，在前面曾提及在计算实例程序的时间花费时，把一些复制内存、配置内存的时间花费也计算在内了，而这些事件花费并不会缩短。实际上 kernel 函数的运行时间大约为 0.053s 左右(性能约相当于 38GFLOPS)，比上一版本又快了将近一倍。

如果此时该实例程序的执行效率已不再受限于内存带宽，那为什么仍未达到预期的效率呢？这是因为除了使用 Kahan's Summation Formula 需要更多的运算外，该实例程序中还存在大量计算矩阵地址的乘法等，都将花费计算资源。另外，那些用于判断是否超出方阵范围的 if 语句，也会对程序的执行效率带来影响。为了去掉 if 语句，在配置内存时，可将其配置为 16 的倍数，并在将矩阵复制到显存之前先将其清零。如下所示：

```
int newn = ((n + BLOCK_SIZE - 1) / BLOCK_SIZE) * BLOCK_SIZE;
cudaMallocPitch((void**) &ac, &pitch_a, sizeof(float) * newn, newn);
cudaMallocPitch((void**) &bc, &pitch_b, sizeof(float) * newn, newn);
cudaMallocPitch((void**) &cc, &pitch_c, sizeof(float) * newn, newn);
cudaMemset(ac, 0, pitch_a * newn); cudaMemset(bc, 0, pitch_b * newn);
```

这时，可去掉 kernel 函数中的 if 判断语句，如下：

```
__global__ static void matMultCUDA(const float* a, size_t lda, const float* b, size_t
```

```
                                        ldb,float * c, size_t ldc, int n)
{
    __shared__ float matA[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float matB[BLOCK_SIZE][BLOCK_SIZE];
    const int tidc = threadIdx.x;
    const int tidr = threadIdx.y;
    const int bidc = blockIdx.x * BLOCK_SIZE;
    const int bidr = blockIdx.y * BLOCK_SIZE;
    int i, j;
    float results = 0;
    float comp = 0;
    for(j = 0; j < n; j += BLOCK_SIZE)
    {
        matA[tidr][tidc] = a[(tidr + bidr) * lda + tidc + j];
        matB[tidr][tidc] = b[(tidr + j) * ldb + tidc + bidc];
        __syncthreads();
        for(i = 0; i < BLOCK_SIZE; i++)
        {
            float t;
            comp -= matA[tidr][i] * matB[i][tidc];
            t = results - comp;
            comp = (t - results) + comp;
            results = t; }
            __syncthreads();
        }
        c[(tidr + bidr) * ldc + tidc + bidc] = results;
    }
```

执行修改后的实例程序，其结果如下所示：

```
CUDA initialized.
Max error: 1.19209e-007 Average error: 4.22751e-008 Time used: 0.0780
Press ENTER to exit...
```

似乎该实例程序的执行效率没有得到明显的改善，不过实际上 kernel 函数的运行时间已减少为 0.042s(执行性能约相当于 48GFLOPS)。

如果将 block 变得更大是否有助于提高该实例程序的执行效率呢？当然，由于此时实例程序已不再受限于内存带宽(在 0.042s 内存取 500MB 数据约相当于 12GB/s 的带宽)，所以将 block 再加大并不会带来性能上的收益。而且，由于一个 block 内最多只能有 512 个 thread，将 block 变大还会带来额外开销。另外，shared memory 的大小也有限制(GeForce 8800GT 的 shared memory 大小被限制在 16 384B 之内)，所以也不能任意增加 block 的大小。

本节小结

>>>

本节主要从 CUDA 常用 API、编程模型、存储器模型等方面介绍了 CUDA 的基本知识。其中，API 和编程模型为 CUDA 程序员提供了 CUDA 的接口。而存储器模型最为重要，在实际应用中，对各级存储器的理解和合理使用会直接影响到 CUDA 程序的执行效率。只有清楚地了解 CUDA 的体系结构，才能最大限度地发挥 GPU 各个部件的性能，才能体现出 GPU 并行计算的优势。

对希望进一步学习 CUDA 的读者，可参考 http://cuda.itpub.net/，该论坛提供了很多 CUDA 程序代码和使用心得。除此之外，还可关注 NVIDIA 公司和 AMD 公司在 GPU 领域中的最近进展。

3.5 Cell BE 上的编程模型与语言

本节将概要介绍 Cell BE 编程的相关知识。在本节中，将重点放在如何使初学者快速入门。其主要内容包括：搭建 Cell BE 编程环境；从头开始编译，运行一个简单的 Cell BE 程序，并了解 Cell BE 程序的基本框架；随后介绍 Cell BE 的相关 API，使读者了解 Cell BE 程序的运行过程；最后介绍如何在 Cell BE 上实现传统矩阵乘的并行化，并简要介绍在模拟器环境下进行 Cell BE 程序的性能分析和优化。

3.5.1 Cell BE 简介

Cell Broadband Engine(Cell BE)处理器是由日本的 Sony、Toshiba 和美国的 IBM 于 2001 年初开始合作研发的一款多核处理器。它基于新的多处理器架构——Cell Broadband Engine Architecture(CBEA)。Sony 于 2006 年下半年发布的新一代游戏主机 PlayStaion3(PS3)就采用了 Cell BE 处理器。Cell BE 的主要设计目标是将 PlayStation2 的处理器性能提高 100 倍。在 2001 年 3 月，这三家公司在美国德州奥斯汀成立了 STI(Sony、Toshiba、IBM)设计中心作为联合开发实验室。在 2005 年，STI 最终完成了 Cell，并发表了一篇技术专利来展示其处理器芯片。在 2006 年 2 月，IBM 还向市场推出了 Cell 刀片计算机系统。Cell 刀片是第一种多内核刀片计算机，为那些运行计算密集型工作和宽带媒体应用带来了突破性的性能提升。

虽然，Cell BE 最初是为多媒体应用(如游戏机和高清电视)而设计，但 Cell BE 并非仅仅是一个专用的处理器，它先进的架构使其非常适用于任何需要在短时间内提供海量计算能力和数据吞吐能力的应用，如 NGN 核心设备、数字信号处理、物理模拟、生物数据分析、高性能商业和科学计算等。在 2005 年 6 月，IBM 和 Mercury Computer Systems 结成合作伙伴关系，共同研发基于 Cell BE 的嵌入式应用，以期将 Cell BE 应用于医疗图像、

工业用途检测、地震数据处理以及电信等领域。Cell BE 正在和其他领域处于领导地位的处理器架构进行激烈的竞争。

Cell BE 处理器与众不同的地方在于：每块芯片包含一个主处理单元(PPE)和另外八个协处理单元(SPE)。这种多核架构是 Cell BE 所特有的。PPE 的作用是运行操作系统，管理系统资源，以及进行控制处理(比如 SPE 线程的申请和管理)。SPE 用于执行 PPE 分配大量数据，运算复杂的子任务等。Cell BE 处理器在设计时便将分布式处理考虑在内，它将高性能的计算任务分成更小的部分，并将这些子任务分配到多个处理单元，每个处理单元都以 4GHz 以上的速度运行。正是这种能力使得 Cell BE 处理器能以 192GFLOPS 的速度运行。另外，基于 Cell BE 的处理器不仅可以在一块芯片上的资源之间执行分布式处理，还可以在多块芯片甚至各个网络设备上执行分布式处理。

3.5.2　第一个 Cell BE 程序

1. SDK 的安装

Cell BE SDK 是专门为在 Cell 体系架构上进行软件和系统开发以及性能分析提供的一个完整的开发工具包。它提供了高效的开发工具和开发库、仿真环境以及大量的技术文档。在 SDK 环境下编译出来的程序，可直接拷贝到实际的 Cell Blade Server 上正常运行。

在安装 SDK 之前，需要完成 Linux 操作系统(Fedora、Ubuntu 均可)的安装，并配置好其 gnome 界面。查看系统是否安装了 rsync、sed、tcl、wget 程序包。否则，使用 yum 工具进行安装，其安装命令为 yum install rsync sed tcl wget。

可在 IBM 的 developworks 网站 http://www-128.ibm.com/developerworks/power/cell/下载 Cell BE SDK 包。该包共有四个文件，其中两个是基本库：cell-install-3.0.0-1.0.noarch.rpm 和 CellSDK-Devel-Fedora_3.0.0.1.0.iso。剩余的两个是扩展库：cell-extras-Fedora-license-3.0.0-2.0.noarch.rpm 和 CellSDK-Extras-Fedora_3.0.0.1.0.iso。

将所有安装文件放入/home/sdk3.1 文件夹，然后按照如下步骤进行安装：

```
$  cd  /home/sdk3.1
$  rpm -ivh cell-install-3.1.0-0.0.noarch.rpm
```

其运行结果如下所示：

```
Preparing...                ########################################## [100%]
   1:cell-install           ########################################## [100%]
```

```
$  cd /opt/cell
$  ./cellsdk   -- iso /home/sdk3.1 install
```

其运行结果如下所示：

```
cellsdk INFO-2050: STARTING cellsdk --iso /home/sdk3.1 install
cellsdk INFO-2041: Copying SDK versions of open source rpms to /tmp/cellsdk/openSrc
cellsdk INFO-2016: Copied 0 of 0 rpms into /tmp/cellsdk/openSrc
cellsdk INFO-2041: Copying SDK versions of open source rpms to /tmp/cellsdk/openSrc
cellsdk INFO-2016: Copied 0 of 0 rpms into /tmp/cellsdk/openSrc
cellsdk INFO-2041: Copying SDK versions of open source rpms to /tmp/cellsdk/openSrc
cellsdk INFO-2016: Copied 0 of 0 rpms into /tmp/cellsdk/openSrc
cellsdk  INFO-2043:  Calling  yum  --disablerepo=*  --enablerepo=CellSDK-Devel-Fedora-x86
--enablerepo=CellSDK-Extras-Fedora-x86 --enablerepo=CellSDK-Open-Fedora-x86 update
Loaded plugins: refresh-packagekit
CellSDK-Extras-Fedora-x86                          | 1.1 kB     00:00
CellSDK-Open-Fedora-x86                            | 1.1 kB     00:00
CellSDK-Devel-Fedora-x86                           | 1.1 kB     00:00
Setting up Update Process
No Packages marked for Update
cellsdk INFO-2046: No rpms need to be installed

cellsdk INFO-2025: All default rpms are installed
cellsdk INFO-2045: The cellsdk install is complete
cellsdk INFO-2051: ENDING cellsdk --iso /home/sdk3.1 install
```

提示安装完成之后，退出安装界面。紧接着安装模拟器，在 console 中运行：

```
$  /opt/cell/cellsdk_sync_simulator install
```

不过，直接这样做可能会有问题。若不能成功安装，可单独下载其模拟器进行安装。下载地址：http://www.alphaworks.ibm.com/tech/cellsystemsimsystemsim-cell-3.1-8.f9.i386.rpm。若安装模拟器时提示缺少 libstd6.so 等库，可以使用 yum 命令进行安装。

当出现"Installation of RPMs into Simulator sysroot image is complete."则表示安装完成。默认只安装必需的 rpm 包。如果还需安装 xlc 和其他 Cell BE 编程的扩展高级库，可手动将两个 iso 文件以命令 mount-o loop 到某个目录下去安装。安装完之后的目录结构如表 3-12 所示。

表 3-12　目录结构

目　录	内　容	备　注
/opt/cell/sdk/src	例子程序	
/opt/cell/sdk/prototype/src	使用 ALF 的例子程序	
/opt/cell/sdk/docs	pdf 文档	包含 Cell BE 编程手册
/opt/ibm/systemsim-cell	与模拟器有关的内容	包含一个小的 Linux 操作系统
/opt/cell/toolchain	编译器和函数库	

2. 运行模拟器

安装完 Cell BE SDK 之后，运行模拟器(如下)：

```
$   export PATH = /opt/ibm/systemsim - cell/bin: $ PATH
$   systemsim -g
```

其运行结果如下所示：

```
GUI Enabled
Licensed Materials - Property of IBM.
(C) Copyright IBM Corporation 2001, 2007
All Rights Reserved.
Using initial run script /opt/ibm/systemsim-cell/bin/../lib/cell/systemsim.tcl
GUI not initialized.    Execute tcl command 'gui_init'.
building tree....
clearing existing Openfirmware tree
done building tree.
LOAD : Opening ELF image file: /opt/ibm/systemsim-cell/bin/../images/cell/vmlinux
Elf text start address saved is 0x0000000001000000
Elf_ReadImage: Opening ELF image file: /opt/ibm/systemsim-cell/images/cell/vmlinux
Elf_ReadImage: alloc-ed 8784792 bytes for /opt/ibm/systemsim-cell/images/cell/vmlinux
vmlinux_get_struct_info got errors: couldn't execute "/opt/ibm/systemsim-cell/bin/../bin/parse_dwarf.pl":
no such file or directory
LOAD : ELF startup: PC=0x0000000001000000, msr=0x1000000000000000
LOAD :                    gpr[1]=0x000000000FFFFF90, gpr[2]=0x0000000000000000
systemsim %
```

模拟器运行之后的界面，如图 3-42 所示。

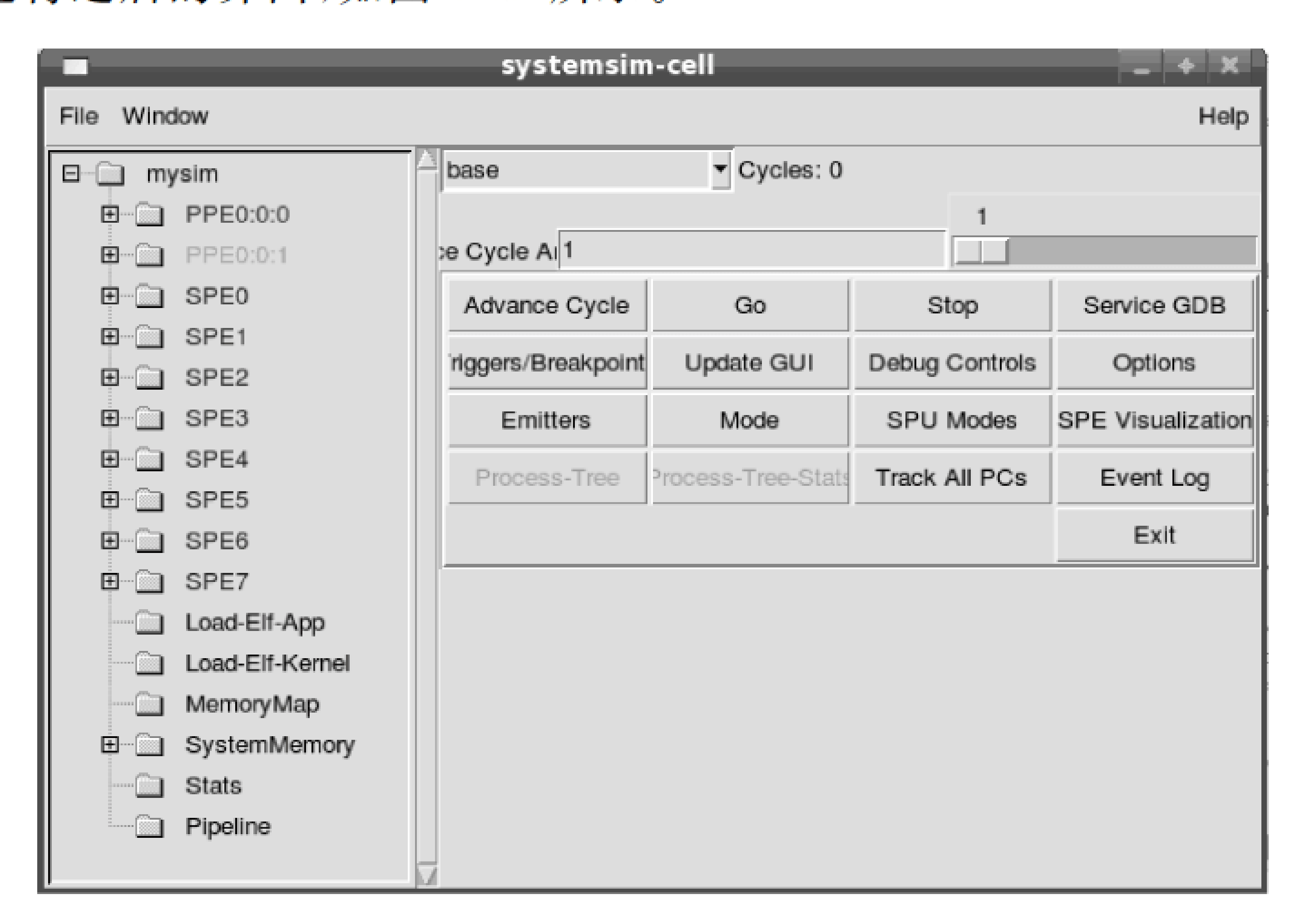


图 3-42　模拟器的运行界面

单击图 3-42 界面上的 Mode，选择 Fast Mode，然后单击 Go 按钮，就会出现启动 Linux 操作系统之后的界面，如图 3-43 所示。

```
ide: Assuming 50MHz system bus speed for PIO modes; override with idebus=xx
mice: PS/2 mouse device common for all mice
platform ppc-rtc.0: rtc core: registered ppc_md as rtc0
usbcore: registered new interface driver hiddev
usbcore: registered new interface driver usbhid
drivers/hid/usbhid/hid-core.c: v2.6:USB HID core driver
TCP cubic registered
Initializing XFRM netlink socket
NET: Registered protocol family 1
NET: Registered protocol family 17
registered taskstats version 1
md: Autodetecting RAID arrays.
md: Scanned 0 and added 0 devices.
md: autorun ...
md: ... autorun DONE.
Initializing disk 0 with devsz 1843200
VFS: Mounted root (ext2 filesystem).
Freeing unused kernel memory: 448k freed
                Welcome to Fedora Fedora release 9 (Sulphur)
                Press 'I' to enter interactive startup.
eth0: bogus network driver initialization
No IRQ retreived
Starting login process
[root@(none) ~]#
```

图 3-43 启动模拟平台的 Linux 系统

3. Cell BE 程序结构

一般情况下，Cell BE 应用程序不会控制 SPE 的物理系统资源，整个系统的所有物理资源都由 Linux 操作系统管理，操作系统会提供某种机制实现应用程序对 SPE 的访问和控制。由应用程序管理和使用的资源称为 SPE 上下文（SPE Context），SPE 上下文是对 SPE 物理系统资源的逻辑描述。

Cell BE SDK 提供 SPE 运行时管理库 libspe 实现应用程序对 SPE 上下文的操作，libspe 是一个标准的底层应用程序编程接口。通过该库及其应用程序接口（API），应用程序可以控制 SPE 的上下文，从而获得访问 SPE 系统资源的机会。SPE 是与操作系统完全不相关的库函数。在实际系统中，SPE 上下文由操作系统调度到物理 SPE 资源上。

在简单的应用程序中，使用 SPE 的基本流程如下：

① 创建一个 SPE 上下文。

② 加载一个 SPE 可执行对象到 SPE 上下文的本地存储器。

③ 运行 SPE 上下文。将控制权交给操作系统，由操作系统实现物理上的上下文调度，并将上下文加载到 SPE 物理资源上。

④ 销毁 SPE 上下文。

上述的第三步需要操作系统的一个同步调用。执行该调用的应用程序会被阻塞，直

到 SPE 停止运行或操作系统从该调用返回。

大多数应用程序需要并发地调用多个 SPE。因此，应用程序必须创建多个线程与并发的 SPE 上下文进行匹配，每个线程一次运行一个 SPE 上下文。例如，如果需要 n 个并发 SPE 上下文，线程数一般为主程序线程加上 n 个 SPE 上下文执行的专用线程。

运行 n 个 SPE 上下文的基本流程如下：

(1) 创建 n 个 SPE 上下文；

(2) 加载合适的 SPE 可执行对象到每个 SPE 上下文的本地存储器中；

(3) 创建 n 个线程(每个 SPE 上下文运行在一个线程上)；

(4) 等待所有 n 个线程停止运行；

(5) 销毁所有 n 个 SPE 上下文。

4. 第一个 Cell BE 程序

创建一个名为 test 的程序进行 Cell BE 的简单编程，该程序使用 PPU 来调度 SPU，计算 4 个整数乘以 4 个浮点数的结果。

一般情况下，Cell BE 程序的目录结构为：

- 一个用于存储整个源代码的工程目录；
- 两个分别存储 PPE 端和 SPE 端的源代码与 Makefile 文件的子目录。

在工程目录下的 Makefile 文件可以调用两个子目录中的 Makefile 进行编译。

下面的 Cell BE 程序以 test 为工程目录。3 个 Makefile 文件和 2 个 c 文件的内容如下：

```
test/Makefile :

DIRS = spu ppu
include /opt/cell/sdk/buildutils/make.footer

test/ppu/Makefile:

PROGRAM_ppu  :=   ../test
IMPORTS = ../spu/libtest_spu.a  - lspe2 -lpthread
include /opt/cell/sdk/buildutils/make.footer

test/spu/Makefile:
PROGRAMS_spu      := test_spu
LIBRARY_embed     := libtest_spu.a
include /opt/cell/sdk/buildutils/make.footer

test/ppu/test_ppu.c:

#include <pthread.h>
```

```
#include <libspe2.h>
/* 定义参与并行运算的 spu 个数(不要超过 8,否则不能真正并行) */
#define SPU_THREADS 2

/* 定义 spe 可执行程序的句柄 */
extern spe_program_handle_t test_spu

/* 线程函数.在 ppu 主程序中创建一个新线程之后,新线程执行此函数 */
void *ppu_pthread_function(void *arg)
{
    spe_context_ptr_t ctx;
    unsigned int entry = SPE_DEFAULT_ENTRY;
    ctx=*((spe_context_ptr_t *)arg);
    /* 运行 spe 程序 */
    if (spe_context_run(ctx, &entry, 0, NULL, NULL, NULL) < 0)
    {
        perror ("Failed running context");
        return NULL;
    }
    /* ppe 线程退出 */
    pthread_exit(NULL);
}

int main ( )
{
    int i;
    spe_context_ptr_t ctxs[SPU_THREADS];
    pthread_t threads[SPU_THREADS];

    for(i = 0; i < SPU_THREADS; i++)
    {
      /* 创建 spe 的上下文 */
        if ((ctxs[i] = spe_context_create (0, NULL)) == NULL)
        {
            perror ("Failed creating context");
            return -1;
        }
      /* 加载 spe 程序 */
       if (spe_program_load (ctxs[i], &test_spu))
        {
            perror ("Failed loading program");
            return -1;
        }
  /* 创建 ppe 线程 */
```

```
        if (pthread_create (&threads[i], NULL, &ppu_pthread_function, &ctxs[i]))

        {
            perror ("Failed creating thread");
            return -1;
        }
    }


    for(i = 0; i < SPU_THREADS; i++)
    {
    /* 等待所有的 ppe 线程结束 */
        if (pthread_join (threads[i], NULL))
        {
            perror("Failed pthread_join");
            return -1;
        }
    /* 销毁 spe 的上下文 */
       spe_contest_destroy(ctxs[i]);
    }

    return (0);
}
test/spu/test_spu.c:
#include <stdio.h>

/* 单个 spu 进行的运算 */
int main()
{
    int a[4] = {140, 270, 450, 822};
    float b[4] = {2, 0.54, 1.78, 0.33};
    int c[4] = {0};
    int i = 0;
    for( i = 0; i<4; i ++ )
    {
            c[i] = a[i] * b[i];
        printf("c[%d] =%d\n", i, c[i]);
    }
    return 0;
}
```

在 test 的根目录下生成一个名为 test 的可执行程序，这个可执行程序(/home/test/test)需要被复制到模拟器环境的 Linux 中才能运行，在模拟器的 console 窗口中输入如图 3-44 所示的命令并获得相应的结果。

```
[root@(none) ~]# callthru  source  /home/test/test > ./test
[root@(none) ~]# chmod +x test
[root@(none) ~]# ./test
c[0] = 280
c[1] = 145
c[2] = 800
c[3] = 271
c[0] = 280
c[1] = 145
c[2] = 800
c[3] = 271
[root@(none) ~]# █
```

图 3-44　在模拟器中运行程序

3.5.3　Cell BE 编程模型简介

在上节程序中，使用了许多的 SPE API。下面，来了解一下这些 API。首先，SPE API 是围绕 SPE 上下文这个概念建立的。SPE 上下文是指 SPE 中完整数据集（包括可执行代码）的当前状态。在 API 中，SPE 上下文的调用是同步的：该调用直到所调用的程序执行完成之后才会结束。主程序代码必须创建多个线程（或进程）来运行多个 SPE。同时，可以直接使用线程 API 来直接控制与线程调度有关的过程。

SPE 程序含有 main 函数，它将被独立编译。SPE 程序简单地执行一些必需的工作，其设置和销毁均由连接到 SPE 程序中的启动代码和 PPE 上运行的库代码自动处理，一般情况下可忽略这些过程。不过，SPE 程序必须做一些自己的数据转换工作。

下面，介绍一下上节程序中出现过的重要数据类型。SPE API 定义了很多不透明的数据类型，它们被用作各个 API 函数的参数。此处不会详细介绍所有这些数据类型的内容，而是简要地介绍在上节程序中用到的几个数据类型。

- spe_context_ptr_t 表示一种虚拟化的 SPE 状态。其中包括了寄存器状态和本地存储中的内容。它是表示程序在 SPE 上执行状态的数据结构，不管程序在被加载、正在运行还是正在查询，或是被停止了，这些状态都可以通过这个数据结构查询。它是一个不透明的句柄。此处不深入介绍其中的内幕。要了解更多的有关内容，请参考文献《Cell BE 处理器编程指南》。
- spe_program_handle_t 是一个句柄，用来标识一个可在 libspe2 中使用的 SPE 可执行程序。该数据类型可在一个包含 SPE 二进制的文件中创建，或使用 ppu-embedspu 工具嵌入 PPE 程序当中。它也是一个不透明的句柄，只由库来使用。
- spe_stop_info_t 用来记录 SPE 程序停止执行的原因。不同于以上两种数据类型，它是一个透明的句柄。其结构在 SDK 文档中有所介绍，其中最重要的成员是 stop_reason，它表示 SPE 程序停止执行的原因。最常见的值是 SPE_EXIT，程序成功执行之后，在该结构中会保存退出的状态。

下面，来看一下上节程序中使用过的 API 函数。

1. spe_context_ptr_t spe_context_create(unsigned int flags, spe_gang_context_ptr_t gang)

- 该函数中的 flag 参数用来设定线程的特性，一般传入 0 即可。
- gang 参数用于将新建的 SPE 上下文与它指向的 spe 组上下文相关联，一般传入 NULL，表示该 SPE 上下文没有与任何 SPE 组上下文相关联。
- 该函数调用成功会返回一个 spe 上下文的指针。

2. int spe_program_load (spe_context_ptr_t spe, spe_program_handle_t * program)

- 该函数中的 SPE 参数指向 spe 程序执行时 SPE 上下文环境，一般会传入 spe_context_create 函数执行后的返回值。
- program 参数指向已经映射到内存的 SPE 端程序。
- 该函数调用成功之后返回 0。

3. int spe_context_run(spe_context_ptr_t spe, unsigned int * entry, unsigned int runflags, void * argp, void * envp, spe_stop_info_t * stopinfo)

- 该函数中的 SPE 参数指向 SPE 程序执行时的 SPE 上下文环境，一般会传入 spe_context_create 函数执行后的返回值。
- entry 参数表示 SPE 程序开始执行时的初始地址，一般传入 SPE_DEFAULT_ENTRY，表示将使用默认的初始地址。
- runflags 参数用来设定 SPE 程序执行时的特性，一般传 0 即可。
- argp 是一个可选的参数，可用作传递给 SPE 程序的第二个参数。
- envp 也是一个可选的参数，可以作为传递给 SPE 程序的第三个参数。
- stopinfo 是一个有效的指针，指向 spe_stop_info_t 结构，SPE 程序执行结束时，此结构体相关信息将被填充，对大多数程序来说，输入 NULL 即可。
- 该函数调用成功之后返回 0 或一个正数。

4. int spe_context_destroy(spe_context_ptr_t spe)

- 该函数中的 SPE 参数指向 SPE 程序执行时的 SPE 上下文，一般会传入 spe_context_create 函数的返回值。
- 该函数调用成功之后返回 0。

从应用程序开发者的角度来看，Cell BE 可被简单地视为一个 9 路多处理器。PPE 是一个基于 PowerPC 架构的双线程双发射、顺序执行的 RISC 处理单元。在一个时钟周期内，它可以处理来自两个线程的指令(即所谓硬件多线程 SMT)。所以，再加上 8 个 SPE，整个处理器能够在同一时刻同时运行 10 个任务。

基本上，PPE 作为控制和任务调度处理器，SPE 则主要处理计算任务。举例来说，在一个 IPTV 机顶盒应用中，需要创建多个 SPE 线程用于视频解码和输出。那么，PPU 负责创建、管理和维护这些 SPE 线程。PPE 是一个通用的 64 位 RISC PowerPC 架构的处理器，因此对 PPE 的编程与普通的 PowerPC 的编程相同，很多在 PowerPC 处理器平台上编写的应用程序可稍作修改甚至不用修改就能移植到全新的 Cell BE 处理器上运行。

在内存访问方式上的不同是 SPE 与 PPE 的一个关键性区别。PPE 通过 load 和 store 指令直接访问主存和寄存器。SPE 则要通过 DMA 访问主存，将指令和数据读取并存入本地存储器，而不是直接共享主存。

为了使 Cell BE 的性能得以充分利用，在 PPE 上运行的操作系统必须能够支持多任务。在 PPE 上运行的主线程负责创建一个或多个 Cell BE 任务。一个 Cell BE 任务有一个或多个主线程和一些 SPE 线程与其相关联。每个 SPE 线程将在一个单独的 SPE 上运行，每个 SPE 都有自己的寄存器。主线程可与 SPE 直接通信或通过主存与 SPE 间接通信。

操作系统会根据一定的策略来调度 SPE 线程。它首先对系统中的 Cell BE 任务进行优先级划分。然后，独立地对 SPE 的执行和主线程进行调度。操作系统还负责完成 SPE 程序的动态加载、参数传递、对 SPE 事件响应以及提供调试支持等。

SPU 只能从本地存储器中取指令和数据。SPU 应用程序使用本地存储器地址，因此 SPU 的 load 和 store 指令只能访问本地存储器。SPE 的 DMA 控制器负责完成本地存储器地址和系统主存地址之间的指令和数据的传送。

Cell BE 有多种编程模型。对于简单的 SPE 应用程序来说，与普通的应用程序编程类似。首先，需要将任务进行划分，针对不同的 SPE 编写不同的代码，每个 SPE 完成一个特定的任务。在这种情况下，SPE 不需要访问主存，只需访问本地存储，SPE 的数据段、代码段的大小不能超过 256KB。

如果数据段和代码段的大小超过了 256KB，则需要使用大型 SPE 编程模式。在该模式下，PPE 会预留一段有效地址空间供 SPE 程序使用，然后将这段有效地址空间的起始地址告知 SPE，SPE 便通过 DMA 方式访问这段内存。

在有些时候，需要多个 SPE 之间进行协同工作，SPE 线程之间就会出现同步问题。在 Cell BE 中，可通过原子操作、信箱机制、SPE 的信号机制、事件和中断机制以及对共享内存的轮询方式实现线程之间的同步。

在多个 SPE 线程协同工作时，存在如下两种工作方式。

① 工作队列方式：在这种方式下，一个空闲的 SPE 将会分配一个任务。

② 流水线方式：每个 SPE 处理同一任务的一部分工作，前一个 SPE 的输出作为后一个 SPE 的输入。

3.5.4　性能分析与优化

1. 在 Cell BE 上实现矩阵乘法

下面的实例通过邮箱和 DMA 在 SPE 和 PPE 之间传递数据以实现矩阵乘法的并行计算。PPE 通知 SPE 待计算矩阵的位置以及各个 SPE 需要计算的行，SPE 计算完毕之后将数据发回给 PPE，PPE 在收集全部数据之后输出结果矩阵。

PPE 上的 matrix_ppu 程序代码如下：

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <libspe2.h>
#include <pthread.h>
#include <libmisc.h>

#define SPU_THREADS 4
#define N 4

void *ppu_pthread_function(void *arg);
void test();

int main()
  {
    test();
    printf("The program has sunccessfully executed.\n");
    return 0;
  }

void *ppu_pthread_function(void *arg)
{
    spe_context_ptr_t ctx;
    unsigned int entry = SPE_DEFAULT_ENTRY;

    ctx = *((spe_context_ptr_t *)arg);
    if(spe_context_run(ctx, &entry, 0, NULL, NULL, NULL)< 0 ) {
         perror("Failed running context");
         exit(1);
    }
    return NULL;
}

void test()
{
    extern spe_program_handle_t simple_spu;
```

```
spe_context_ptr_t ctxs[SPU_THREADS];
pthread_t threads[SPU_THREADS];
int ma[N][N] __attribute__((aligned(128)));
int mb[N][N] __attribute__((aligned(128)));
int result[N][N] __attribute__((aligned(128)));

srand((unsigned)time(NULL));
unsigned int i,j;
for(i = 0; i < N; i++)
    for(j = 0; j < N; j++)
    {
        ma[i][j] = rand() % 100;
        mb[i][j] = rand() % 100;
        result[i][j] = 0;
    }

for(i = 0; i < SPU_THREADS; i++)
  {
    /* 创建 SPE 上下文 */
    if((ctxs[i] = spe_context_create(0, NULL)) == NULL)
    {
        perror("Failed creating context");
        exit(1);
    }
    /* 加载 SPE 端程序到上下文 */
    if(spe_program_load(ctxs[i], &simple_spu))
    {
        perror("Failed loading program");
        exit(1);
    }
    /* 创建线程 */
    if(pthread_create(&threads[i], NULL, &ppu_pthread_function, ctxs + i))
    {
        perror("Failed creating thread");
        exit(1);
    }
}

/* 将要计算的行通过邮箱传递给 SPE */
for(i = 0; i < SPU_THREADS; i++)
{
    spe_in_mbox_write(ctxs[i], &i, 1, SPE_MBOX_ANY_NONBLOCKING);
}
```

```
/* 将要计算的矩阵地址通过邮箱传递给 SPE */
unsigned int pma, res;
unsigned int pmb = (unsigned int)mb;
for(i = 0; i < SPU_THREADS; i++)
{
    pma = (unsigned int)&ma[i][0];
    res = (unsigned int)&result[i][0];
    spe_in_mbox_write(ctxs[i], &pma, 1, SPE_MBOX_ANY_NONBLOCKING);
    spe_in_mbox_write(ctxs[i], &pmb, 1, SPE_MBOX_ANY_NONBLOCKING);
    spe_in_mbox_write(ctxs[i], &res, 1, SPE_MBOX_ANY_NONBLOCKING);
 }

unsigned int dataFin;
for(i = 0; i < SPU_THREADS; i++)
 {
    dataFin = 0;
    while(spe_out_mbox_status(ctxs[i]) < 1);
    spe_out_mbox_read(ctxs[i], &dataFin, 1);
    if(dataFin == 1)
        continue;
}

/* SPE 端计算完成之后通知 PPE,并将结果数据传回 PPE */
unsigned int taskFinished = 0;
for(i = 0; i < SPU_THREADS; i++)
{
    while(spe_out_mbox_status(ctxs[i]) < 1);
    spe_out_mbox_read(ctxs[i], &taskFinished, 1);
     if(taskFinished == 2)
     {
         taskFinished = 0;
     }
}

printf("Matrix A is:\n");
for(i = 0; i < N; i++)
{
    for(j = 0; j < N; j++)
    {
        printf(" %d  ", ma[i][j]);
     }
    printf("\n");
}
printf("Matrix B is:\n");
```

```
    for(i = 0; i < N; i++)
   {
        for(j = 0; j < N; j++)
        {
            printf(" %d  ", mb[i][j]);
         }
        printf("\n");
    }

    printf("Matrix A multiply matrix B is:\n");
    for(i = 0; i < N; i++)
    {
        for(j = 0; j < N; j++)
        {
            printf(" %d  ", result[i][j]);
        }
        printf("\n");
    }

    /* 等待所有的 PPE 线程结束,销毁 SPE 上下文 */
    for(i = 0; i < SPU_THREADS; i++)
    {
        if(pthread_join(threads[i], NULL))
        {
            perror("Failed pthread_join");
            exit(1);
        }
    }
}
```

SPE 上的 matrix_spu 程序代码如下：

```
#include <stdio.h>
#include <spu_mfcio.h>

#define N 4
void cellComput();
int myRow[N] __attribute__ ((aligned(128)));
int res[N] __attribute__ ((aligned(128)));
int mb[N][N] __attribute__ ((aligned(128)));

int main()
{
```

```
    int i;
    for(i = 0; i < N; i++)
        res[i] = 0;
    unsigned int speNumber;
    speNumber = spu_read_in_mbox();
    /* 获取待计算矩阵的地址 */
    unsigned int ma_addr, mb_addr, res_addr;
    ma_addr = spu_read_in_mbox();
    mb_addr = spu_read_in_mbox();
    res_addr = spu_read_in_mbox();

    /* 获取数据 */
    mfc_get(myRow, ma_addr, 16, 0, 0, 0);
    mfc_write_tag_mask(1);
    mfc_read_tag_status_all();

    mfc_get(&mb[0][0], mb_addr, 64, 0, 0, 0);
    mfc_write_tag_mask(1);
    mfc_read_tag_status_all();

    /* 发出数据已接收完毕的信号 */
    spu_write_out_mbox(1);
    cellComput();
    /* 将计算结果传回 PPE */
    mfc_put(res, res_addr, 16, 0 , 0, 0);
    mfc_write_tag_mask(1);
    mfc_read_tag_status_all();
    /* 计算结束 */
    spu_write_out_mbox(2);
    return 0;
}

void cellComput()
{
    int k, j;
    for(j = 0; j < N; j++)
        for(k = 0; k < N; k++)
        {
            res[j] += myRow[k] * mb[k][j];
        }
}
```

该实例程序的运行结果如下所示：

```
[root@ps7 tom]# ./simple
Matrix A is:
22  26  9  34
53  63  48  32
0  40  41  47
75  34  40  60
Matrix B is:
8  41  85  92
83  66  81  80
71  72  94  81
0  22  33  62
Matrix A multiply matrix B is:
2973  4014  5944  6941
9061  10491  15176  15788
6231  6626  8645  9435
6262  9519  14869  16580
The program has been sunccessfully executed.
```

注意，由于该实例中的矩阵都是随机生成的，所以每次运行的结果会有所不同。

2. 简要性能分析

模拟器可精确计算出 SPU 程序运行所花的 CPU Cycle 数，据此可对 Cell BE 程序进行性能调优。如果需要模拟器精确统计出 SPU 程序运行所花的 CPU Cycle 数的话，单击模拟器上的 Mode 按钮，然后选择 Cycle Models，如图 3-45 所示进行模拟器的配置。

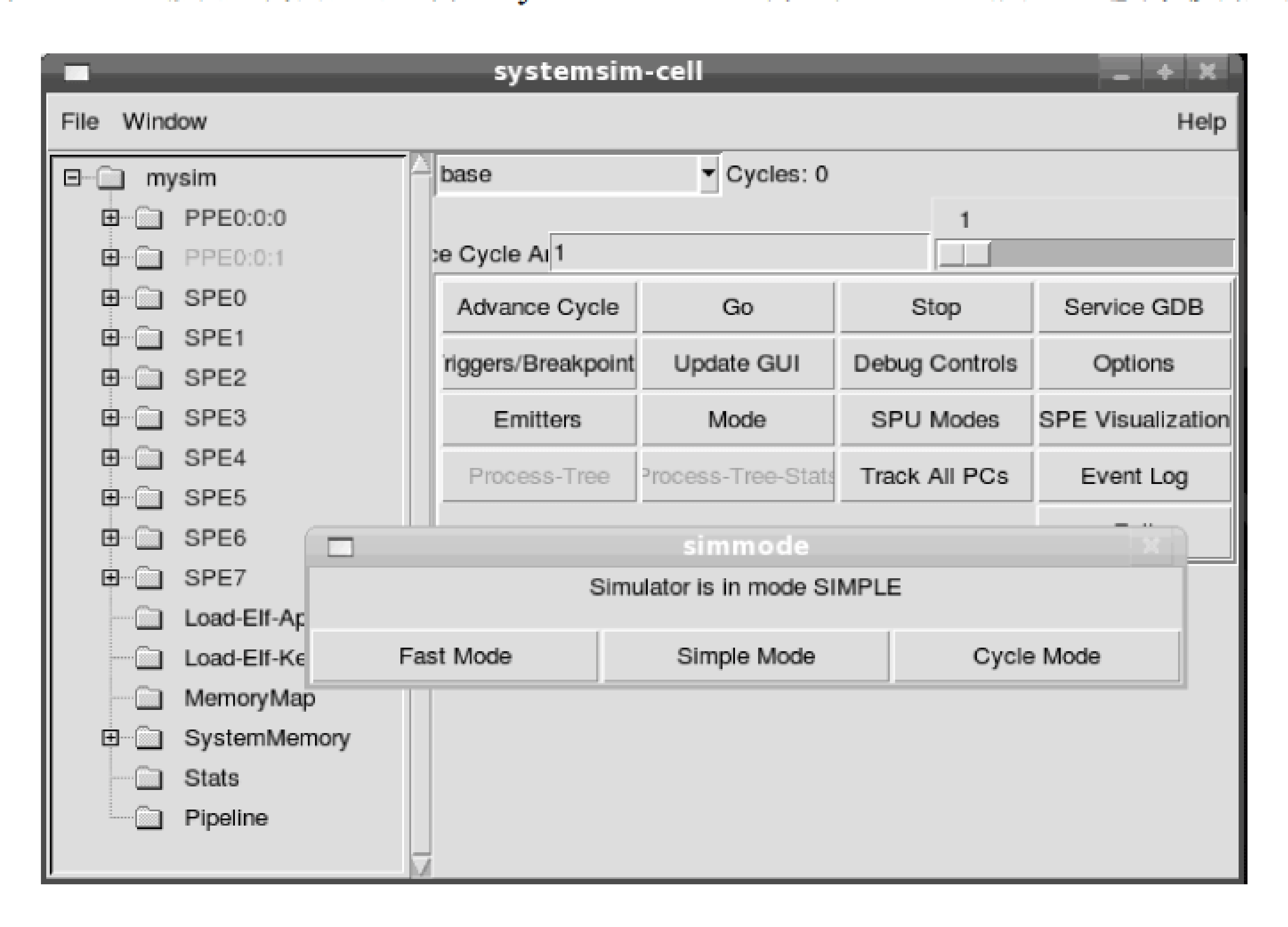


图 3-45　SPU Mode 配置

配置完成之后，关闭弹出的选项窗口。然后，如图 3-44 所示，在 console 中运行 test 程序，在模拟器的左栏中即可看见相关的性能数据。

尽管在模拟器上可以看到程序运行较详细的性能数据，但是一般需要看到的是程序在真实 Cell BE 平台上的运行情况。在上述矩阵乘法程序中加上时间函数，可以测得在

Cell BE 平台上两个 4×4 矩阵乘大约花费时间在 45μs 左右。时间测试程序运行得到结果如下：

```
[root@ps7 tom]# ./time_simple
Program's  running time is: 0.000045 seconds
```

3. 优化 Cell BE 应用程序

因为 PPE 是一个通用的 64 位 RISC PowerPC 架构的处理器，所以已有的 PowerPC 应用程序经重新编译之后，一般情况下均能在 Cell BE 上顺利地执行。然而，该方式仅仅利用了 PPU，却未充分发挥 SPU 的计算能力。

设置一些编译选项之后，Cell BE 编译器就能对应用程序进行一定程度的自动优化。目前，Cell BE 的 C/C++编译器和开发库对 C 和 C++做了大量的扩展，程序员要充分利用 Cell BE 的多处理器架构与 SIMD(Single Instruction Multiple Data)能力，进行应用程序的优化。

程序员可在代码中使用 SIMD 指令和数据结构，规定编译器进行哪些方面的代码优化。SIMD 化是进行的主要优化。经测试表明，一般情况下，代码经过 SIMD 化之后，其执行性能都得到了大幅提高。利用并行计算的思想和算法(对计算任务进行划分，再把各个任务分配到不同的 SPU 上)，对应用程序的性能提高同样有效。如果按照并行编程来设计一个应用程序，可以减少指令流水线的等待和停顿，同样可以提高程序的执行性能。

下面是 Cell BE 编程手册中的一个例子，来看看它是如何对代码进行 SIMD 优化的。其中，array_sum 是一个对数组进行求和的函数。

```
int array_sum(unsigned char nums[16])
{
  int sum = 0;
  int i;
  for (i = 0; i < 16; ++i)
    {
      sum += nums[i];
    }
  returnsum;
}
```

如上所示，array_sum 是一个普通的求和函数。它通过循环遍历数组，对数组的每个元素进行求和。在 ps 上运行这个程序，测得求和时间如下所示：

```
[root@ps7 tom]#./time_sum
Time of sum is 96 ns.
```

下面，利用 Cell BE 编译器对 SIMD 指令的支持，来消除上述代码中不必要的串行循环，从而提高代码的运行效率。改进后的代码(函数 array_sum 对应函数 vectorized_

sum)如下：

```
union
{
  int s[4];
  vector signed int v;
} sum;
int vectorized_sum(unsigned char nums[16])
{
  vector unsigned char v_nums;
  vector unsigned int zero = (vector unsigned int){0};
  v_nums = vec_perm(vec_ld(0, nums), vec_ld(16, nums), vec_lvsl(0, nums));
  sum.v = vec_sums((vector signed int)vec_sum4s(v_nums, zero), (vector signed int) zero);
  return (sum.s[3]);
}
```

改进之后，函数 vec_perm 从偏移地址 0～16，以左对齐的方式，读入 nums 中的 16 个字节，构成 vector。vec_sums 则是对两个 vector 进行求和。此处，vec_sum4s 将变量 vnums 按四路并行的方式与一个值为全 0 的 vector 进行向量求和，即相当于对 nums 的元素进行逐个相加。在 ps 上运行这个程序，测得求和时间如下所示：

```
[root@ps7 tom]# ./time_vec_sum
Time of sum is 73 ns.
```

可见，经过优化后的数组求和时间减少了 23 ns，比原来性能提高了约 24%。

本节小结

本节主要介绍了初学者进行 Cell BE 编程需要掌握的基本内容，主要包括：搭建 Cell BE 编程环境，了解 Cell BE 程序的基本框架和运行过程，以及 Cell BE 程序并行化与优化的相关知识。本节虽然对与 Cell BE 编程有关的更深层次的知识（包括 DMA、通信与异步、向量编程等高阶内容）有所涉及，但读者如果希望进一步了解 Cell BE 编程的相关知识的话，可深入学习 Cell BE 编程手册等资料。

参考文献

1 张武生，薛巍等. MPI 并行程序设计实例教程. 北京：清华大学出版社，2009.
2 都志辉. 高性能计算并行编译技术——MPI 并行程序设计. 北京：清华大学出版社，2001.
3 莫则尧，袁兴国. 消息传递并行编程环境 MPI. 北京：科学出版社，2001.
4 国科技大学微固体结构研究室 MPI 教程. http://micro.ustc.edu.cn/Linux/MPI/MPICH/.
5 Kurt Wall 著，张辉译. GNU/Linux 编程指南(第二版). 北京：清华大学出版社，2002.
6 百度百科 gcc 词条. http://baike.baidu.com/view/4848.html.

7 维基百科 gcc 词条. http://zh. wikipedia. org/wiki/GCC.
8 The GNU OpenUP Implementation,http://gcc. gnu. org/onlinedocs/gcc-4. 6. 0/libgorup. pdf
9 OpenMP 官方网站. http://www. openmp. org/ OpenMP.
10 Barbara Chapman, Gabriele Jost, et al. Using OpenMP. Cambridge Massachusetts: the MIT press,2008.
11 陈文光,武永卫. MPI 与 OpenMP 并行程序设计. 北京:清华大学出版社,2004.
12 陈国良,安红等. 并行算法实践. 北京:高等教育出版社,2004.
13 陆鑫达等译. 并行程序设计原理. 北京: 机械工业出版社, 2009.
14 J Dean,S Ghemawat. MapReduce:Simplified Data Processing On Large Clusters. Communications of the ACM,2008,51(1):107～113.
15 S Ghemawat,H Gobioff,S k Leung. The Google File System. Operating Systems Review,2003,37(5):29～43.
16 Apache Lucene Overview . http://lucene. apache. org/java/docs/index. html.
17 D Borthakur. The Hadoop Distributed File System:Architecture and Design. http://hadoop. apache. org/common/docs/r0. 16. 0/hdfs_design. html.
18 HBase Overview. http://hbase. apache. org/.
19 Fa y Chang, Dean J, et al. Bigtable: a distributed storage system for structured data. ACM Transactions on Computer Systems,2008,26(2):26～29.
20 NVIDIA 官方网站. http://www. nvidia. com.
21 NVIDIA Corporation. NVIDIA CUDA Compute Unified Device Architecture Progrmming Guide. http://developer. download. nvidia. com/computer/cud/1_0/NVIDIA_CUDA_programming_Guide_2. 0. pdf.
22 Garland,Le Grand,et al. Parallel Computing Experiences with CUDA. IEEE Micro,2008,28(4):13～27.
23 赵开勇博客. http://blog. csdn. net/OpenHero.
24 张舒,褚艳丽等. GPU 高性能运算之 CUDA. 北京: 中国水利水电出版社. 2009.
25 CUDA 开发者论坛. http://cuda. itpub. net/.
26 Cell BE 编程模型. http:// servers. pconline. com. cn/skills/0706/1043004. html.
27 Cell BE 百度百科. http:// baike. baidu. com/view/3136215. htm.
28 IBM 官方网站. http:// www. ibm. com/developerworks/cn/linux/l-cn-cellprogramming/.
29 Cell BE 编程模型. http:// servers. pconline. com. cn/skills/0706/1043004_2. html.
30 林海波, 谢海波等. Cell BE 处理器编程指南. 北京: 电子工业出版社, 2008.
31 Cell BE 官方网站. http:// www. alphaworks. ibm. com/topics/cell.

CHAPTER 4 第4章 并行应用实例——大规模稀疏线性方程组求解的并行化

4.1 稀疏线性方程组及其求解方法

4.1.1 稀疏线性方程组的应用

随着应用规模的不断扩大与计算机技术的快速发展，科学计算在计算机应用领域中占据着越来越重要的地位。在科学与工程计算领域(包括计算化学、油藏模拟、地震资料处理、数值天气预报、电力系统仿真设计以及高维方程数值求解等)中，线性方程组的求解尤其是大规模稀疏线性方程组的求解处于核心的地位。

大量的实验结果表明，在许多问题的求解过程中，线性方程组求解所花的时间会占到总的执行时间的一半以上，从而成为整个问题求解的瓶颈。由于稀疏矩阵包含了大量的零元素，因此需要设计专门的存储格式与特定算法，才能满足高效求解此类问题的需求。本章主要介绍大规模稀疏线性方程组求解的迭代算法，并对一个计算案例的并行化进行详细分析。

4.1.2 大规模稀疏线性方程组求解的迭代算法

对于大规模稀疏线性方程组求解，通常的方法有直接法和迭代法这两种。其中，直接法是通过对稀疏矩阵进行分解计算求解来获得方程组的解，其主要的问题是在分解过程中会引入大量的填入元，从而增加存储的开销，同时，直接法的算法复杂度也比较高。而迭代法则通过不断地循环迭代，当方程组的解满足收敛条件时，终止循环。对于大多数的问题来说，迭代法能够在较少的迭代次数情况下满足收敛要求。而且，一般来说，迭代法的存储开销也比直接法的存储开销小。采用迭代法时，方程组求解的收敛性依赖于矩阵本身的特性。对于有些问题，迭代法可能不收敛，而对于条件数较好的数值问题，迭代法却能快速地收敛。目前，对大规模稀疏线性方程组求解，迭代法已成为主流计算方法。

4.1.3 Krylov 子空间迭代法

1. Krylov 子空间概念

Krylov 子空间迭代法是一种用于求解形如 $Ax=b$ 的方法，从而解决一些大规模线性

方程组的求解问题。

对于给定线性方程组 $\boldsymbol{A}x=b$(其中,A 是 $n\times n$ 的矩阵),定义 m 维 Krylov 子空间为:

$$K_m(\boldsymbol{A},r) = \operatorname{span}\{r,\boldsymbol{A}r,\boldsymbol{A}^2r,\cdots,\boldsymbol{A}^{m-1}r\}$$

给定初始的 $x(0)$,在 m 维空间 K(右子空间)中寻找 x 的近似解 $x^{(1)}$ 满足残向量 $r=b-\boldsymbol{A}x^{(1)}$ 与 m 维空间 L(左子空间)正交,即 $b-\boldsymbol{A}x^{(1)}\perp L$,把该条件称为 Petrov-Galerkin 条件。

Krylov 子空间迭代法就是将求解 Ax=b 的过程转化为如下的迭代格式来进行求解:

$$Kx^{(i+1)} = Kx^{(i)}+b-\boldsymbol{A}x^{(i)}$$

其中,K 是构造的近似于 $\boldsymbol{A}$ 的矩阵,其主要的思路就是将一个较为复杂的问题逐步转换为可以求解的子问题。

2. Krylov 子空间迭代法分类

根据 K_m 和 L 的不同,可将 Krylov 子空间迭代法进行分类,主要有如下四类(参考自文献[2]~[3])。

1) 正交投影方法

当 $L=K_m(\boldsymbol{A}, r)$ 时,称这类 Krylov 子空间迭代法为正交投影法。Hestenes 和 Stiefe 提出的共轭梯度法(CG)是其中最重要的方法之一,它要求 A 为对称正定矩阵。CG 的算法流程如图 4-1 所示。

```
Compute r0 : = b − Ax0 . p0 : = r0
For j = 0,1,...,until convergence Do:
    αj : = (rj ,rj )/(Apj ,pj )
    xj+1 : = xj + αj pj
    rj+1 : = rj − αj Apj
    βj : = (rj+1 ,rj+1 )/(rj ,rj )
    pj+1 : = rj+1 + βj pj
EndDo
```

图 4-1　正交投影法(CG)的算法流程

除 CG 之以外,全正交化法(FOM)以及正交残差法(ORTHORES)也属于正交投影法,不过这两种方法不要求 $\boldsymbol{A}$ 为对称正定矩阵。

2) 正交化方法

当 $L=\boldsymbol{A}K_m(\boldsymbol{A},r)$时,称这类 Krylov 子空间迭代法为正交化法。Saad 提出的广义极小残差法(GMRES)是这类方法的典型代表。这类方法的使用范围广泛,因此一直是人们研究的重点,有许多的变形方法。GMRES 的算法流程如图 4-2 所示。

```
Compute r0 = b − Ax0, β: = ‖r0‖2, and v1 : = r0/β
For j = 1,2,...,m Do:
    Compute wj : = Avj
    For i = 1...j Do:
        hij : = (wj, vi)
        wj : = wj − hij vi
    EndDo
    hj+1,j = ‖wj‖2. If hj+1,j = 0 set m: = j and go to 11
    vj+1 = wj/hj+1,j
EndDo
Define the (m+1) × m Hessenberg matrix H̄m = {hij}1≤i≤m+1,1≤j≤m.
Compute ym the minimizer of ‖βe1 − H̄m y‖2 and xm = x0 + Vm ym
```

图 4-2 正交化法(GMRES)的算法流程

共轭余差(Conjugate Residual,CR)法及其推广形式 GCR(Generalized Conjugate Residual)法也属于正交化方法。正交化法具有极小残差的特性,但是随着迭代次数的不断增加,其计算量和存储需求也会随之增加,因此,在实际的计算过程中往往需要采用重启动技术。

3) 双正交化法

当 $L=K_m(A^T,r)$时,称这类 Krylov 子空间迭代法为双正交化法。这类方法主要用于 A 为非对称的情况下。Lanczos 提出的双共轭梯度法(BiCG)是其中最基本的方法。BiCG 的算法流程如图 4-3 所示。

```
Compute r0 : = b − Ax0. Choose r0* such that(r0, r0*) ≠ 0.
Set. p0 : = r0, p0* : = r0*
For j = 0,1,...,until convergence Do:
    αj : = (rj, rj*)/(Apj, pj*)
    xj+1 : = xj + αj pj
    rj+1 : = rj − αj Apj
    r*j+1 : = rj* − αj A^T pj*
    βj : = (rj+1, r*j+1)/(rj, rj*)
    pj+1 : = rj+1 + βj pj
    p*j+1 : = r*j+1 + βj pj*
EndDo
```

图 4-3 双正交化法(GMRES)的算法流程

BiCG 的计算要用到 $\mathbf{A}^T$,而且收敛性不佳。为了避免这些不足之处,人们在此基础上提出了一些其他方法,如共轭梯度平方法(CGS)、广义共轭梯度平方法(GCGS)、共轭梯度稳定性法(BiCGSTAB)以及逆最小残差法(QMR)等。

4) 法方程组法

当 $L=K_m(\mathbf{A}^T\mathbf{A},\mathbf{A}^T r)$时,称这类 Krylov 子空间迭代法为法方程组法。这类方法的主

要思想是将 CG 法应用于求解法方程组 $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}x=\boldsymbol{A}^{\mathrm{T}}b$ 或 $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}u=b,x=\boldsymbol{A}^{\mathrm{T}}u$。其中,CGNR 和 CGNE 是这类方法的典型算法。CGNR 与 CGNE 的算法流程如图 4-4 与图 4-5 所示。

Compute $r_0 = b - Ax_0$, $z_0 = A^T r_0$, $p_0 = z_0$.
For $i = 0, \ldots$, until convergence Do:
    $w_i = Ap_i$
    $\alpha_i = \| z_i \|^2 / \| w_i \|_2^2$
    $x_{i+1} = x_i + \alpha_i p_i$
    $r_{i+1} = r_i - \alpha_i w_i$
    $z_{i+1} = A^T r_{i+1}$
    $\beta_i = \| z_{i+1} \|_2^2 / \| z_i \|_2^2$.
    $p_{i+1} = z_{i+1} + \beta_i p_i$
EndDo

图 4-4 法方程组法(CGNR)的算法流程

Compute $r_0 = b - Ax_0$, $p_0 = A^T r_0$.
For $i = 0, 1, \ldots$, until convergence Do:
    $\alpha_i = (r_i, r_i)/(p_i, p_i)$
    $x_{i+1} = x_i + \alpha_i p_i$
    $r_{i+1} = r_i - \alpha_i Ap_i$
    $\beta_i = (r_{i+1}, r_{i+1})/(r_i, r_i)$
    $p_{i+1} = A^T r_{i+1} + \beta_i p_i$
EndDo

图 4-5 法方程组法(CGNE)的算法流程

4.1.4 预处理技术简介

1. ILU 预处理

一般地,预处理就是将线性方程 $\boldsymbol{A}x=b$ 转化为其等价形式 $\boldsymbol{M}^{-1}\boldsymbol{A}x=\boldsymbol{M}^{-1}b$ 的过程。其中,$\boldsymbol{M}$ 是对 $\boldsymbol{A}$ 的近似,被称为预条件子。通过选择 $\boldsymbol{M}$,可使预处理之后的线性方程比原线性方程更易于采用迭代法进行求解。

预条件的方法有很多,对于不同的应用通常会采用不同的预条件子。比较常见的一种方法是不完全 LU 分解(incomplete LU factorization,ILU)。该方法的主要思想是对原线性方程的系数矩阵 $\boldsymbol{A}$ 进行分解,即 $\boldsymbol{A}=\boldsymbol{L}\boldsymbol{U}-\mathrm{R}$。其中,$\boldsymbol{L}$ 和 $\boldsymbol{U}$ 分别是下三角矩阵和上三角矩阵,$\boldsymbol{R}$ 是剩余矩阵。通过预先指定哪些位置的元素为 0,从而限制填入元(见参考文献[1])。基本的 ILU 算法流程如图 4-6 所示(见参考文献[3]),其中的 P 是定义元素为 0 的模式集合(zero pattern set)。

```
For each(i,j)∈P set αij = 0
For k = 1,…,n − 1 Do:
    For i = k + 1,n and if(i,k)∉P Do:
        αik : = αik /αkk
        For j = k + 1,…,n and for(i,j)∉ P Do:
            αij : = αij − αik * αkj
        EndDo
    EndDo
EndDo
```

图 4-6 基本的 ILU 算法流程

除了最基本的 ILU(0)预处理之外,还有其他变化形式。例如(见参考文献[4]):

- 多层填充的不完全分解预条件 ILU(p),其主要目的是为了弥补 ILU(0)预处理的不足,可在不完全分解因子中采用填入元。此时,一般有两种填充方法:一种基于矩阵结构分析来进行填充;另一种是基于对分解因子中元素值大小的判定来进行填充;
- 增加阈值的 ILU 预条件,即当在分解过程中某些元素的值小于设定的阈值时丢弃。

2. 稀疏近似逆预条件

稀疏近似逆是一个稀疏矩阵 $\boldsymbol{M}$,它一般是对稀疏矩阵 $\boldsymbol{A}$ 的逆 $\boldsymbol{A}^{-1}$ 的一个较好的近似。稀疏近似逆的主要优势在于它们可以并行计算,具有良好的并行性。其主要思想是:矩阵 $\boldsymbol{M}$ 可通过某种方式被构造出来,预条件过程是一个矩阵向量运算且可被并行化。这种类型的预条件子可以求解一些 ILU 预处理难以解决的问题。

当前,稀疏近似逆预条件技术大致有三种(见参考文献[5]):基于 Frobenius 范数极小化、分解的稀疏近似逆、基于 ILU 分解计算的稀疏近似逆。每种预条件子都有一些不同的构造方式且每种方式都有其优缺点,需要根据实际问题的特点采用合适的方法。

4.2 大规模稀疏线性方程组求解案例

4.2.1 Helmholtz 方程及其计算特征

某个气象模式对大气的动力框架最终被转化为一个 Helmholtz 方程,求解 Helmholtz 方程就是求解一个大规模稀疏线性方程组。以全球 25km 算例为例,其规模达到了 1440×720×36=37 324 800 维。

在全球三维网格离散化之后，每个内点涉及到 19 个离散点，其具体格式如图 4-7 所示。

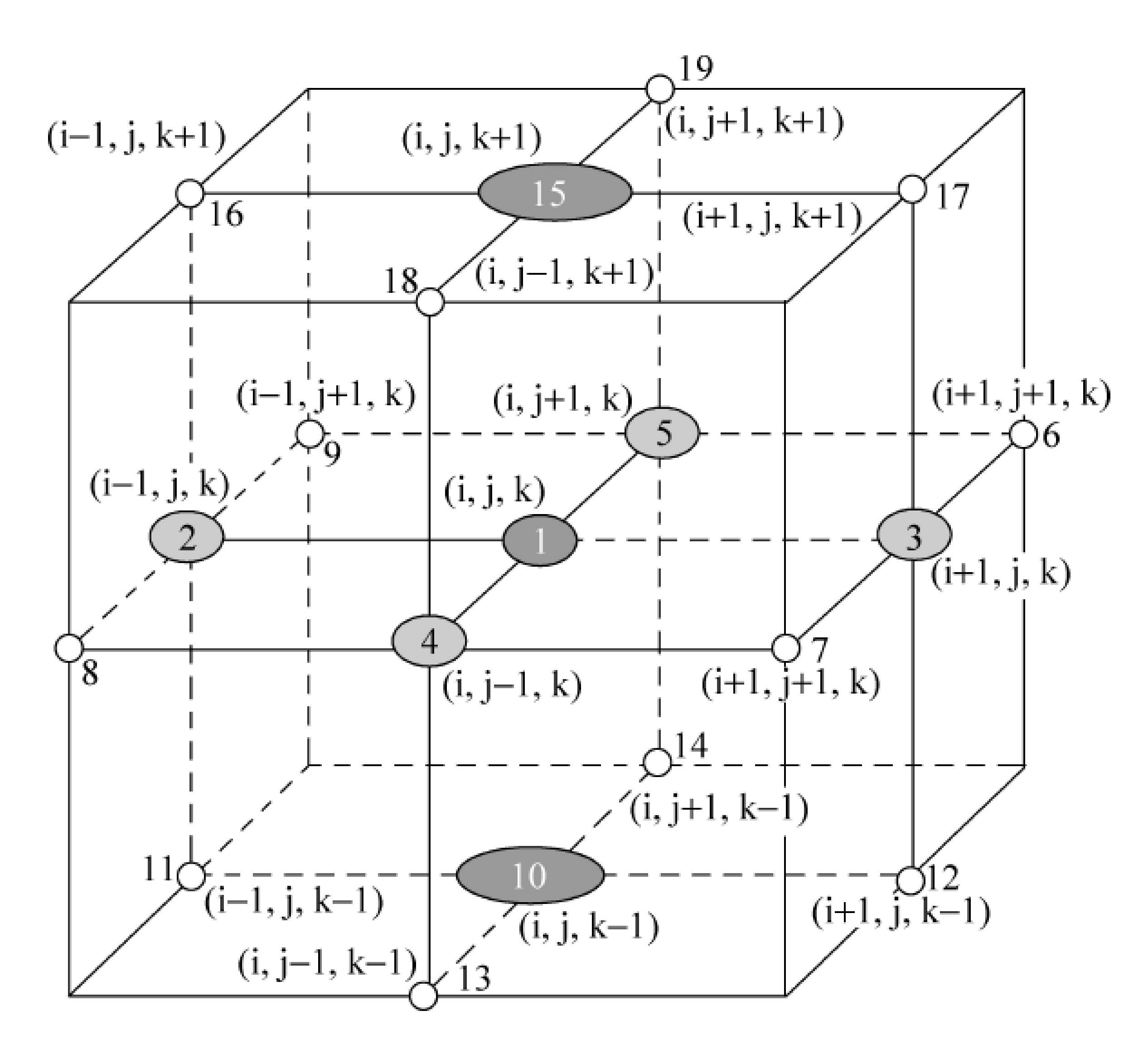


图 4-7　三维网格的空间离散关系

对于其中的某一点(i, j, k)，Helmholtz 方程的形式如下：

$$
\begin{aligned}
& C_1(\prod)_{i,j,k} + C_2(\prod)_{i-1,j,k} + C_3(\prod)_{i+1,j,k} + C_4(\prod)_{i,j-1,k} + C_5(\prod)_{i,j+1,k} \\
& + C_6(\prod)_{i+1,j+1,k} + C_7(\prod)_{i+1,j-1,k} + C_8(\prod)_{i-1,j-1,k} + C_9(\prod)_{i-1,j+1,k} + C_{10}(\prod)_{i,j,k-1} \\
& + C_{11}(\prod)_{i-1,j,k-1} + C_{12}(\prod)_{i+1,j,k-1} + C_{13}(\prod)_{i,j-1,k-1} + C_{14}(\prod)_{i,j+1,k-1} \\
& + C_{15}(\prod)_{i,j,k+1} + C_{16}(\prod)_{i-1,j,k+1} + C_{17}(\prod)_{i+1,j,k+1} + C_{18}(\prod)_{i,j-1,k+1} + C_{19}(\prod)_{i,j+1,k+1} \\
& = (\xi_{n_0})_{i,j,k}
\end{aligned}
$$

在上式中，一共有 19 个系数。对变量按照坐标进行自然排序就可得到一个多对角、稀疏、非对称的线性方程组：

$$\boldsymbol{A}x = b$$

其中，$\boldsymbol{A}$ 为非对称矩阵，$\boldsymbol{A}$ 中每行最多有 19 个非零元素，网格自然排序之后最终形成 19 对角矩阵，$\boldsymbol{A}$ 具有对角占优特性。

以全球 25km 的算例为例，经过对角规格化之后，$C_1 = 1.0$，C_{10} 和 C_{15} 在 10^{-1} 量级，C_2 和 C_3 在 10^{-2} 量级，C_4 和 C_5 在 10^{-3} 量级，而其他值均小于 10^{-5}。值得一提的是，在 Helmholtz 方程求解过程中，$\boldsymbol{A}$ 矩阵不变，只有右端项随时间变化。

4.2.2 Helmholtz 方程的求解

1. GCR 算法

目前，Helmholtz 方程的求解采用 GCR(广义共轭余差)算法，属于 krylov 子空间迭代法。采用 GCR 算法的理由是其收敛速度较快，且实现较为容易。计算流程如图 4-8 所示(见参考文献[3])。

Compute $r_0 = b - Ax_0$. Set $p_0 = r_0$

For $j = 0, 1, \ldots,$ until convergence Do:

$$\alpha_j = \frac{(r_j, Ap_j)}{(Ap_j, Ap_j)}$$

$$x_{j+1} = x_j + \alpha_j p_j$$

$$r_{j+1} = r_j - \alpha_j Ap_j$$

Compute $\beta_{ij} = -\dfrac{(Ar_{j+1}, Ap_i)}{(Ap_i, Ap_i)}$, for $i = 0, 1, \ldots, j$

$$p_{j+1} = r_{j+1} + \sum_{i=0}^{j} \beta_{ij} p_i$$

EndDo

图 4-8 GCR 的算法流程

GCR 算法在数学上与经典的 GMRES 算法等价，从其算法流程中可以看到，需要存储 p_i 和 Ap_i，且每一迭代步的运算操作代价比 GMRES 高 50%。但它相对于 GMRES 的实现更为简单，且易于并行实现。与 GMRES 类似，它也可以采用重启动的方法。

2. ILU 预处理

在 4.1.4 节中对一般的 ILU 预处理做了介绍。特别地，对于有限差分方程，可将模版(stencil)定义为差分网格的一个局部基本单元。在离散差分方程中，与模版中心各点直接相关的格点都被包含在模版中。针对这一问题，由三维 Helmholtz 方程导出十九对角形式的矩阵 $\boldsymbol{A}$，对其进行 ILU(0)分解，其中的矩阵 $\boldsymbol{L}$ 和 $\boldsymbol{U}$ 可通过递推关系获得。

在 ILU(0)的方案中，分解之后的 $\boldsymbol{L}$ 和 $\boldsymbol{U}$ 的乘积与 $\boldsymbol{A}$ 相比，在所有非零位置处均相等，生成的填入元被忽略。从变量求解的顺序上来看，该方案实际上是沿着主对角线方向的两条“波浪”来依次交替得到上三角矩阵和下三角矩阵的系数。

然而，一些实验结果却表明，采用十九对角形式的预条件虽然可以构造更好的预条件，以使 GCR 迭代的次数降低。但是，十九对角的矩阵 ILU(0)分解过程十分复杂，而且生成 ILU 和使用 ILU 的时间也会增加。因此，有必要寻找 ILU 预处理与迭代收敛之间的平衡，使得总的计算时间达到最优。

通过对原矩阵十九对角系数的分析，可以发现垂直层中间点(C_1, C_{10}, C_{15})上的系数的数量级为 $1 \sim 10^{-1}$，(C_2, C_3, C_4, C_5)上的系数约为 $10^{-3} \sim 10^{-5}$，其他结点上的系数为

10^{-7}。忽略值较小的系数之后，将其保留为七对角形式。这种预条件虽然相对粗糙，但是实现起来较为容易，而且也便于得到不同精度的 ILU(k) 预条件子。为了避免通信开销，ILU 预处理过程只在分块的内部进行。

3. 实际采用的计算模式

通过对迭代算法和预处理的分析，得到如下实际采用的带预处理的 GCR(k) 算法。其中，k 为重启动值，$\boldsymbol{M}$ 为预处理矩阵。带预处理的 GCR(k) 算法流程如图 4-9 所示。

1　Compute $R_0 = b - Ax_0, \hat{R}_0 = M^{-1}R_0, p_0 = \hat{R}_0$
2　Do $i = 1, 2, \ldots,$ until convergence
3　　$\alpha_{i-1} = \langle R_{i-1}, Ap_{i-1}\rangle / \langle Ap_{i-1}, Ap_{i-1}\rangle$
4　　$x_i = x_{i-1} + \alpha_{i-1} p_{i-1}$
5　　$R_i = R_{i-1} - \alpha_{i-1} Ap_{i-1}$
6　　$\hat{R}_i = M^{-1} R_i$
7　　Do $j = \mathrm{int}[(i-1)/k]k, \ldots, i-1$
8　　　$\beta_{ij} = -\langle A\hat{R}_i, Ap_j\rangle / \langle Ap_j, Ap_j\rangle$
9　　EndDo
10　　$p_i = \hat{R}_i + \sum_{j=\mathrm{int}[(i-1)/k]k}^{i-1} \beta_{ij} p_j$
11　　$Ap_i = A\hat{R}_i + \sum_{j=\mathrm{int}[(i-1)/k]k}^{i-1} \beta_{ij} Ap_j$
12　EndDo

图 4-9　带预处理的 GCR(k) 算法流程

在实现过程中，上述带预处理的 GCR(k) 算法在 step2～step12 中需要进行多次重复迭代，每次迭代需要完成的计算包括：

- 2 次稀疏矩阵向量乘(SpMV)(其中，step3 和 step8 各进行一次)；
- 1 次针对七对角预处理矩阵的前代回代计算(step6 运算操作，该部分主要是应用预处理条件，利用前代回代来实现。在此之前的生成过程中，将七对角预处理矩阵进行近似分解，产生上三角矩阵和下三角矩阵)；
- 最多 m＋3 次向量内积计算。其中，m 为循环生成的子空间维度(step3 有 2 次向量内积计算。在循环过程中，用于校验误差 d 时需要 1 次向量内积，step7 和 step8 最多需要 m 次向量内积计算)；
- 最多 m＋2 次向量乘加操作(step4 和 step5 有 2 次向量乘加操作，step10 最多有 m 次向量乘加操作)。

GCR 的并行计算与串行计算的不同之处在于计算区域的不同。GCR 在并行计算时，每个进程仅对自己所负责的求解区(数据区、求解区和 halo 区的定义详见后续的图 4-11)进行上述计算即可。但为了求解区计算的正确性，计算需要涉及到 halo 区中数据的通信和访问操作。以全球 25km 的算例(并行度为 1024 时)为例，单分区的计算量约为 3.5M 个乘加操作，而且随着进程数目的增加该计算量还可能进一步降低。

4.3 Helmholtz 方程计算的并行化

4.3.1 并行性分析

对 Helmholtz 方程进行水平两维划分，如图 4-10 所示。

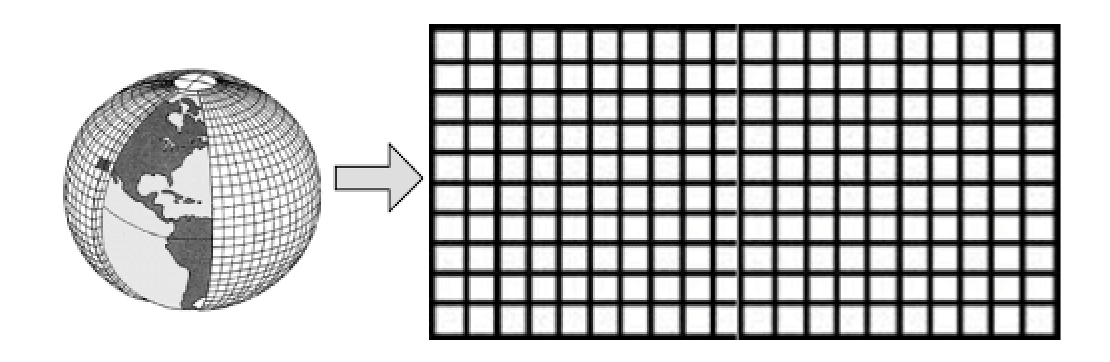


图 4-10 Helmholtz 方程的水平两维划分示意图

在 Helmholtz 方程的并行计算中，每个子区域(非边界区域)都与上、下、左、右四个子区域相关(见参考文献[6])。处理器结点仅计算本结点上的数据空间(求解区)，但在模式的计算过程中可能需要用到其他处理器结点上的变量值，因此通常会在本机上存储部分相邻处理器的变量值，使得在单个处理器上变量的存储空间(数据区)一般要比计算空间大。变量存储空间中与其他处理器结点相交的部分就是通常所说的重叠区域(halo 区)。如图 4-11 所示，由于各处理机的计算不重叠，在计算过程中可能导致本机 halo 区中的值与邻近处理机的计算值不一致，因此在必要时需要对 halo 区进行同步，即需要对 halo 区的变量值进行通信。目前，在 Helmholtz 方程中，halo 区的大小通常采用 1 圈。

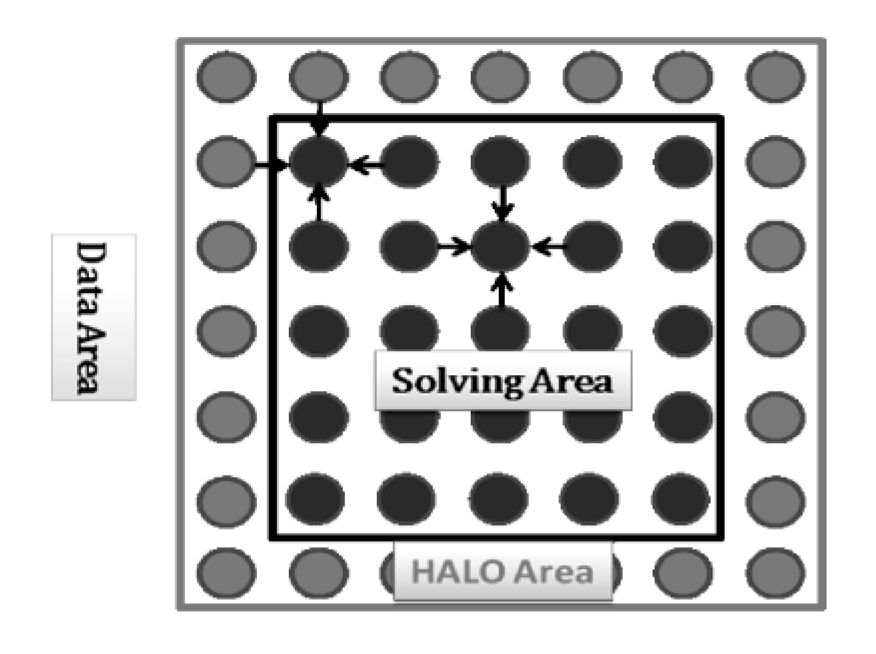


图 4-11 Helmholtz 方程并行计算中的子区域数据空间示意图

4.3.2 通信模式

在划分计算区域的过程中，会引入通信开销。而且通信开销占整个计算代价的比重很大，因此需要合理设计通信模式。Helmholtz 方程的求解主要涉及到两类通信问题，即全局通信和邻居通信。

1. 全局通信

通过对算法流程的分析可知，在 step3 和 step8，还包括检查收敛条件，均需内积运算，而内积运算是对每个进程的计算结果求和并将所得到的计算结果传回每个进程。

MPI_Allreduce 函数可完成下面的操作：组内所有进程都被作为根，分别执行一次规约操作，操作完毕之后所有进程接收缓冲区的数据均相同。该操作等价于首先进行一次 MPI_Reduce 操作然后再执行一次 MPI_Bcast 操作。

对于每一次的循环迭代，均需两次全局通信（Allreduce）。消息的长度是 m+4 个 Double 数据（其中，m 从 1 到 k 循环），总的消息长度约为 100 字节。

2. 邻居通信

除了全局通信，在 GCR 算法中，还采用了邻居通信模式。该模式主要用于边界数据的更新，即重叠区的数据交换。在极点区域，采用的是 MPI 的 send 和 recv 函数，其他区域则采用 sendrecv 函数。

两次边界数据交换（一次是 step6 之后需要重新更新 halo 区的数据，另一次是在 step8 中 SpMV 操作之后需要重新更新周围数据）。

在消息长度上，当 halo 区的数值为 1 时，则需要交换的数据有 2×(I+2)×sizeof(float)+2 ×(J+2)×sizeof(float)，总共进行四次数据交换，涉及到周围相关的 8 个进程。(其中，I 为经度方向上的网格大小，J 为纬度方向上的网格大小)

在通信模式上，GCR 算法以近邻通信为主。而在极区引入轴对称通信，在边界区引入循环通信，其具体的通信拓扑如图 4-12 所示。

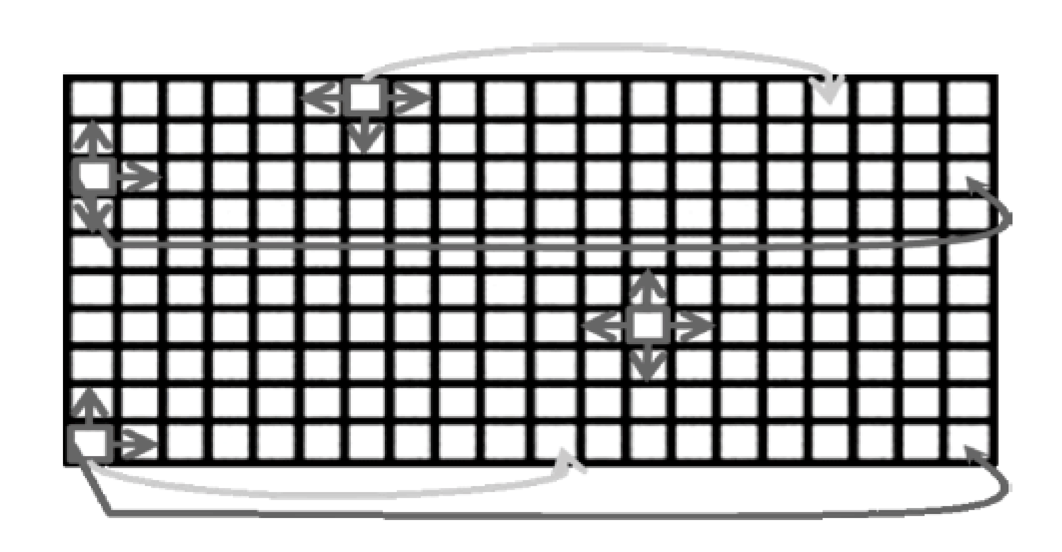


图 4-12　某个气象模式中 GCR 算法的通信模式

在通信频次和消息大小上，通过四次邻居通信，与邻近的 8 个进程完成了边界数据的传入与传出。以全球 25km 的算例（并行度为 1024 时）为例，其通信量约为 4KB。

在 GCR 算法中，邻居通信模式的具体实现过程如下所示。

1）在东西方向上交换数据（参见图 4-13）

由于在东西方向上其邻居通信可看成一个环，因此引入循环通信。包括如下两个步骤：

- 数据从西到东（sendrecv）；
- 数据从东到西（sendrecv）。

2）在南北方向上交换数据(参见图 4-14)

非极区：

- 数据从南到北(sendrecv)；
- 数据从北到南(sendrecv)。

极区：

- 接收和发送非极区边界的相邻数据。需要注意 send 和 recv 的顺序(南极与北极的这两个操作正好相反)，否则会导致死锁。
- 在极区内部，采用轴对称通信，即以极点为中心的对称数据交换，如图 4-12 中黄色箭头所示。

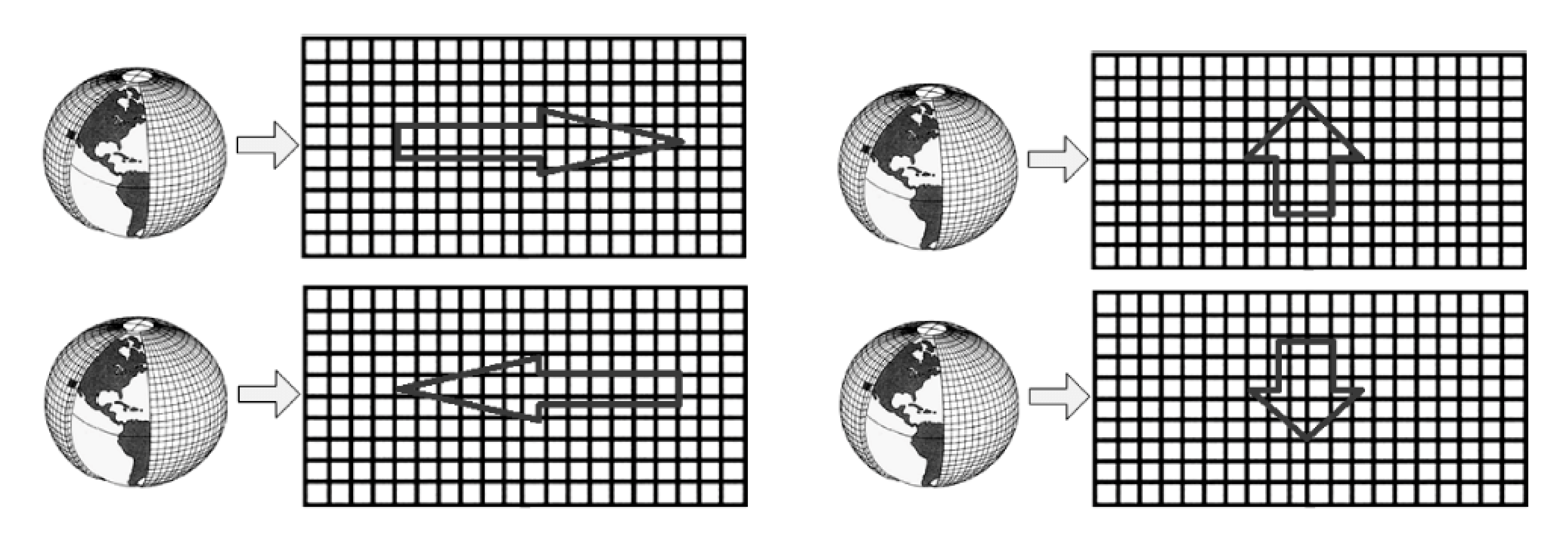


图 4-13　在东西方向上交换数据的示意图　　图 4-14　在南北方向上非极区交换数据的示意图

4.4　实际测试结果与性能优化

4.4.1　测试环境与测试用例

将某个气象模式在天河一号上进行测试，天河一号的配置情况如下：

- 每个计算结点集成了 2 个 Intel X5670 2.93GHz 的六核处理器；
- 每个服务结点包含 2 个 Intel EP CPU，32G 内存；
- 互连通信子系统为两级 Infiniband QDR 互连，单个通信链路的通信带宽为 40Gb/s，延迟 1.2us；
- I/O 存储子系统为全局分布共享的并行 I/O 结构，Lustre 并行文件系统。

对不同的矩阵规模进行实际测试，测试用例为矩阵维度分别为 720×360×36 和 1440×720×36。

4.4.2 测试结果及其分析

1. 原算法测试结果及其分析

在上述测试系统上，对两种分辨率的数据分别进行了测试。图中，横坐标为进程数，纵坐标为加速比。

当矩阵维数 720×360×36 时，实际测试结果如图 4-15 所示。

当矩阵维数为 1440×720×36 时，实际测试结果如图 4-16 所示。

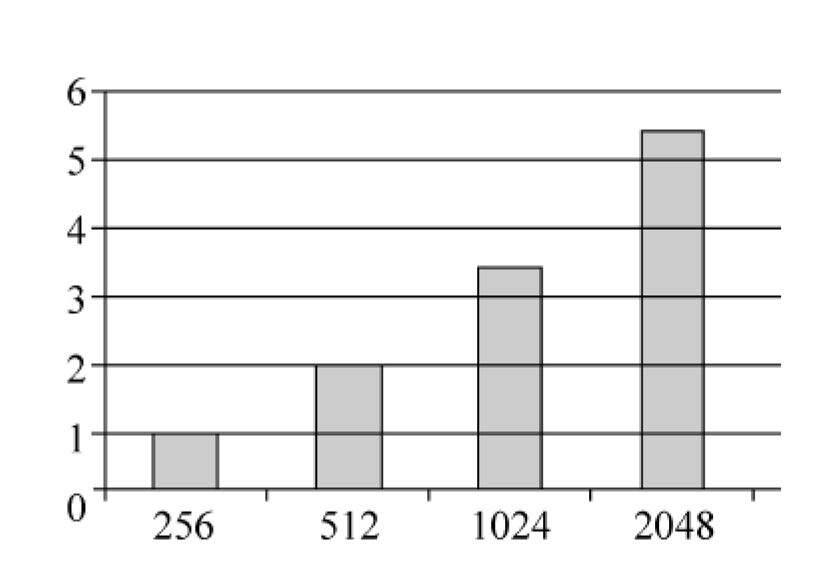


图 4-15 Helmholtz 方程 50 公里分辨率时的加速比

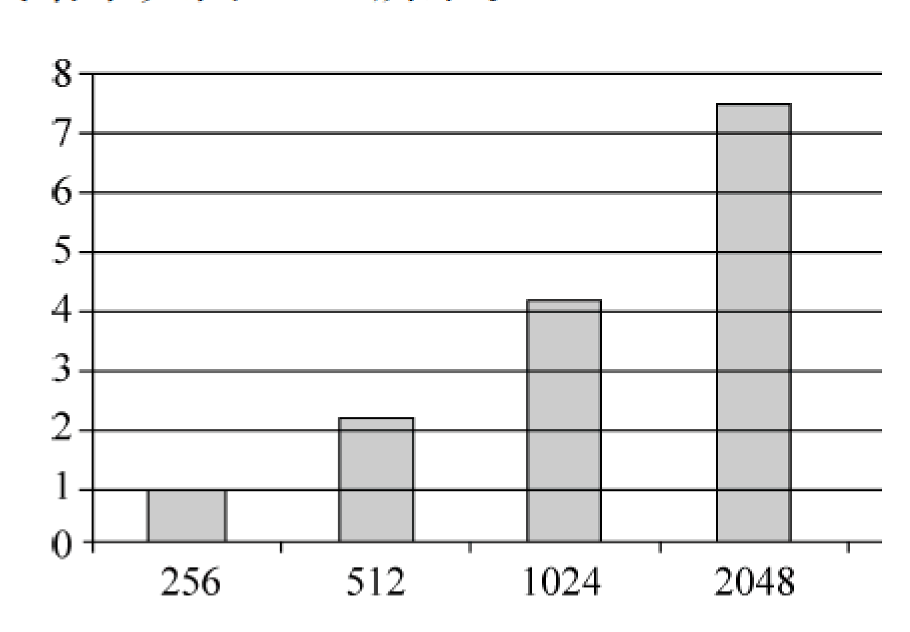


图 4-16 Helmholtz 方程 25 公里分辨率时的加速比

从上述的实际测试结果可以看出，Helmholtz 方程的并行求解具有很好的可扩展性，能够满足并行求解大规模稀疏线性方程组的要求。

2. 优化预处理技术

对预处理进行了如下优化，以实现原有算法的性能改进。

1) 优化局部预处理

将局部预处理从 ILU(0)修改为 ILU(2)，即采用更高级别的预处理，考虑了一部分分解产生的填入元，以提高计算收敛性，降低通信次数及其相应开销。

采用的 ILU(2)属于算法 ILU(p)中的一种，一般形式的 ILU(p)算法如图 4-17 所示。

```
For all nonzero elements aij,define lev(aij) = 0
For i = 2,…,n,Do
  For each k = 1,…,i - 1 and for lev(aik)< = p,Do
    Compute aik = aik/akk
    Compute ai * = ai * - aikak *
    Update the levels of fill of the nonzero ai,j's using levij = min{levij,levik + levkj + 1}
  EndDo
    Replace any element in row i with lev(aij)>p with zero
  EndDo
```

图 4-17 一般形式的 ILU(p)的算法流程

其中，每个元素 aij 对应的 levij 初始化定义如下：

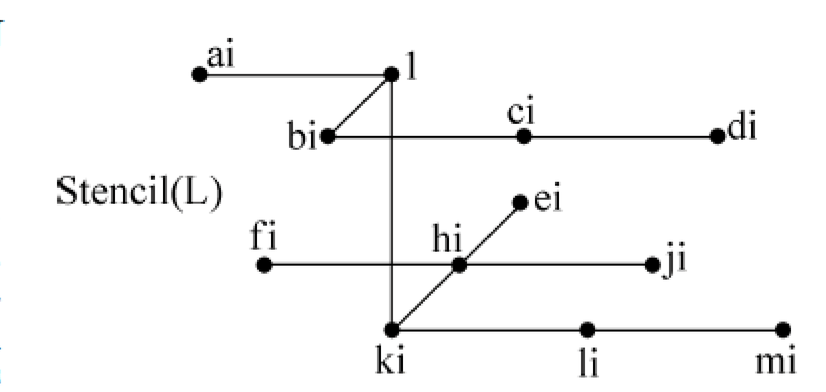
$$levij=\begin{cases}0 & \text{if } aij\neq 0 \text{ or } i=j\\ \infty & \text{otherwise}\end{cases}$$

与十九对角形式的预处理 ILU(0)相类似，也可使用模板来定义格点系数。七对角形式的预处理 ILU(2)之后 L 和 U 的模板及系数如图 4-18 所示。

可通过如下公式来计算：

$$\begin{aligned}stencil_i(LU) = & 1\times stencil_i(U) + ai\times stencil_{i-1}(U) + b_i\times stencil_{i-m}(U)\\ & + c_i\times stencil_{i-m+1}(U) + d_i\times stencil_{i-m+2}(U) + e_i\times stencil_{i-mn+2m}(U)\\ & + f_i\times stencil_{i-mn+m-1}(U) + h_i\times stencil_{i-mn+m}(U) + j_i\times stencil_{i-mn+m+1}(U)\\ & + k_i\times stencil_{i-mn}(U) + l_i\times stencil_{i-mn+1}(U) + m_i\times stencil_{i-mn+2}(U)\end{aligned}$$

简单来说，ILU(2)相对于原来的七对角 ILU(0)来说，考虑了一些生成的元素，即：模版 L 中的系数 c、d、e、f、h、j、l、m 以及模版 U 中的系数 s、r、p、t、q、o、v、w。该算法使得 LU 分解以后得到的近似矩阵与原矩阵更为接近，预处理效果更好。使用公式中系数的计算过程也可依次对变量进行求解。



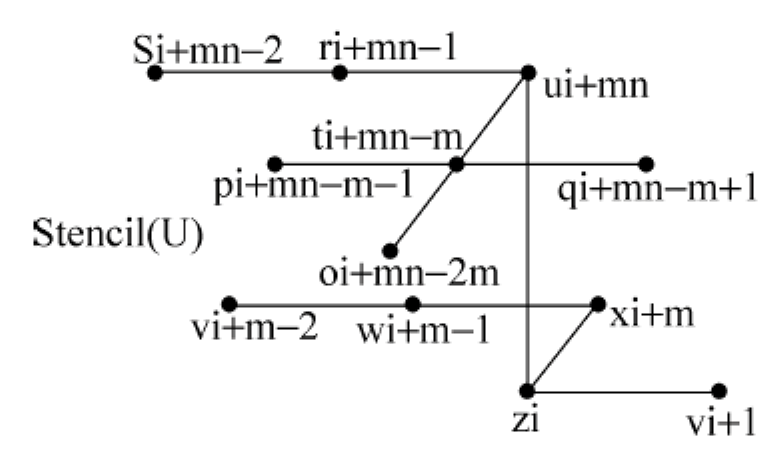


图 4-18 七对角形式的 ILU(2)预处理之后的 L 和 U 模版及系数

2）限制型 Additive-Schwarz 预处理

原预处理采用的是局部预处理，即在划分的区域内部进行预处理。其好处在于不必考虑全局通信，缺点是割裂了划分块与块之间的关系，影响到预处理的效果。

为了改进原有的局部预处理，采用基于限制型 Additive-Schwarz 预处理的重叠预处理策略。例如，可以考虑如图 4-19 所示的二维划分，在经度和纬度方向上的进程数目均为 2。

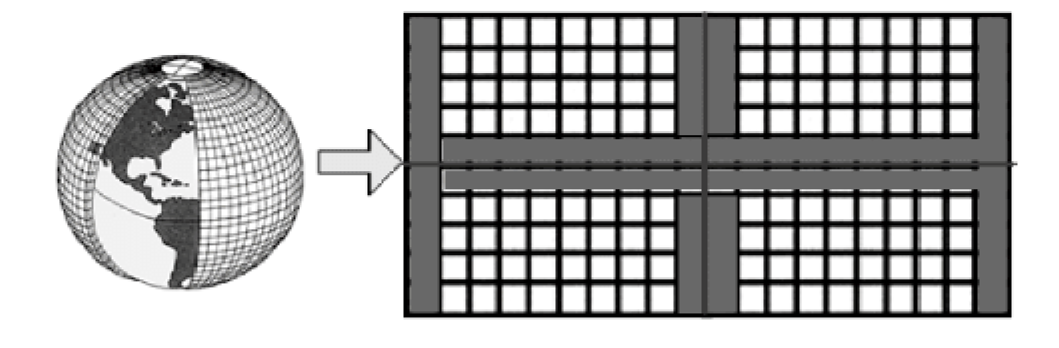


图 4-19 重叠预处理改进示意图

在图 4-19 中，红色部分表示：针对预处理矩阵，每个进程都会在除自己本部分数据之外，还需要获取周围重叠区域的数据，并对该带有重叠区的矩阵进行预处理分解。由于限制型策略仅仅需要更新本区域内的解向量，因此进一步减少了 1 次 updatehalo 的通信开销。

对方程进行了36个时步的迭代计算，预处理改进前后迭代次数之间的比较如图4-20所示。

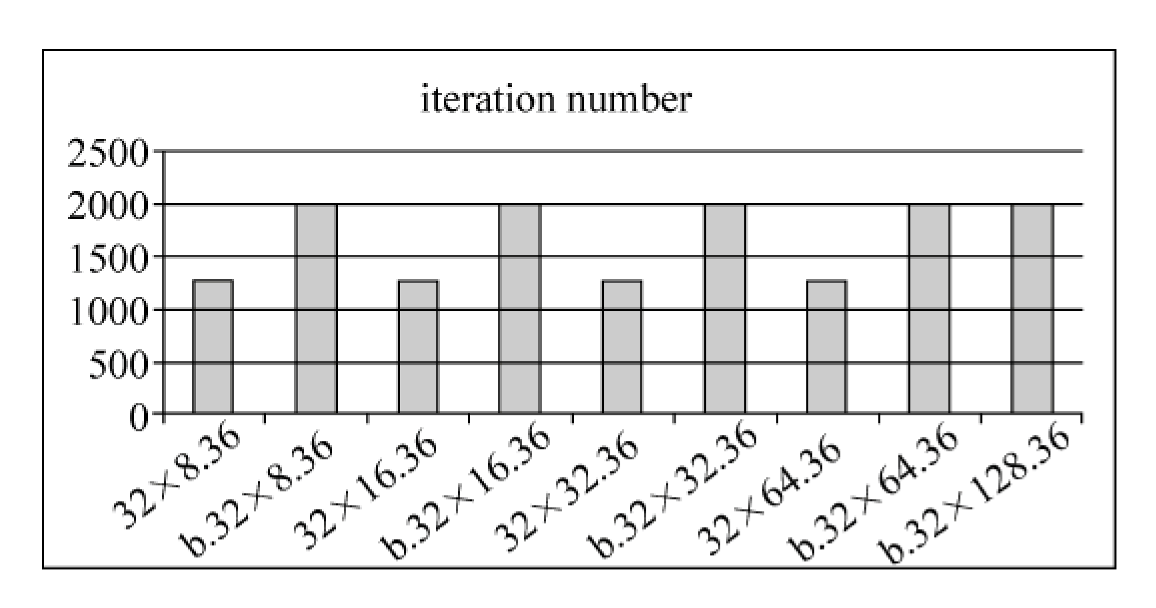


图 4-20　预处理改进前后迭代次数之间的比较

在出现“错误！未找到引用源”时，带有“b.”的为原预处理的迭代次数，不带“b.”的为改进之后预处理的迭代次数。可以看出，改进之后预处理的迭代次数比原预处理的迭代次数减少了将近1/3。不仅如此，GCR整个计算过程的并行加速比也得到了明显的提高，如图4-21所示。

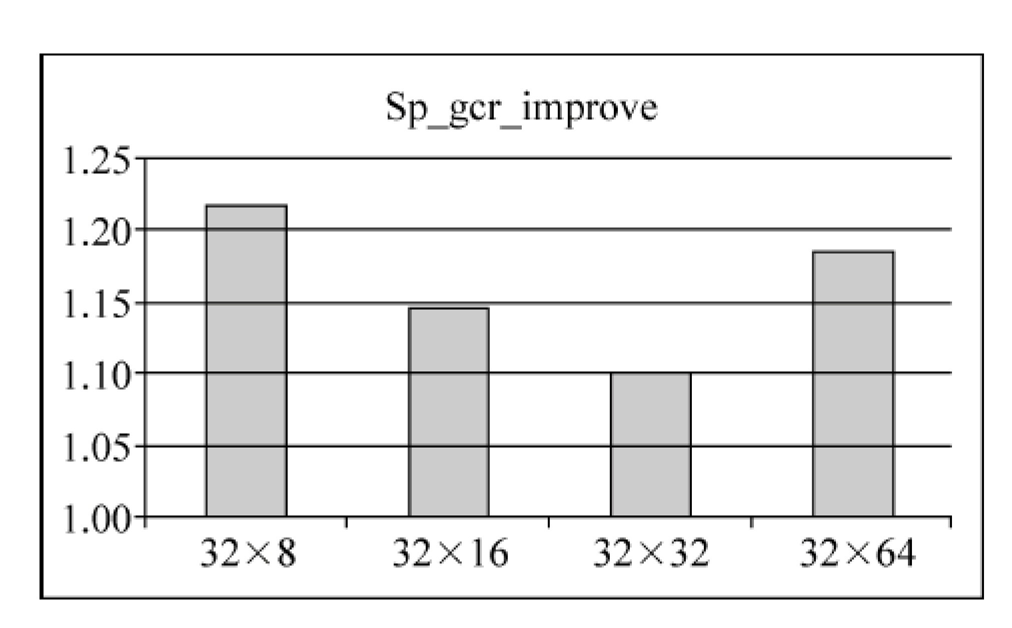
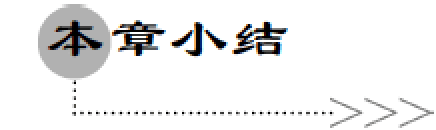


图 4-21　预处理改进之后 GCR 的并行加速比



本章主要介绍大规模稀疏线性方程组求解及其并行化过程。首先介绍了迭代算法与预处理技术。然后，以在气象领域中求解Helmholtz方程为例，给出其并行化方案，并对此方案进行了实际测试与结果分析。在原算法的基础上，对算法实现的预处理部分进行了优化，明显地提高了GCR整个计算过程的并行加速比。

参考文献

1　刘宇，曹建文．适用于GRAPES数值天气预报软件的ILU预条件子．计算机工程与设计，2008，29(3)：731-734.

2 李晓梅，吴建平．Krylov 子空间方法及其并行计算．计算机科学，2005，32(1)：19-20＋40.
3 Saad Y. Iterative methods for sparse linear system. 2nd ed. Philadelphia. PA:SIMA,2003.
4 李晓梅,吴建平．稀疏线性方程组不完全分解预条件方法.计算机工程与科学，2006，28(8)：59-62.
5 谷同祥,迟学斌,刘兴平．稀疏近似逆与多层块 ILU 预条件技术．http://downlunwen.zdnet.com.cn/H049736/pdf/H049736065.pdf.
6 伍湘君，金之雁等．新一代数值预报模式 GRAPES 的并行计算方案设计与实现．计算机研究与发展．2007，44(3)：510-515.